NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A PERFORMANCE ANALYSIS OF VIEW MATERIALIZATION STRATEGIES FOR SELECT-PROJECT-JOIN EXPRESSIONS

by

Jesse T. South

September, 1991

Thesis Advisor: Magdi N. Kamel

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (*If applicable*) 37	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION 8b OFFICE SYMBOL (*If applicable*)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (*City, State, and ZIP Code*)	10 SOURCE OF FUNDING NUMBERS Program Element No / Project No / Task No / Work Unit Accession Number

11 TITLE (*Include Security Classification*)
A PERFORMANCE ANALYSIS OF VIEW MATERIALIZATION STRATEGIES FOR SELECT-PROJECT-JOIN EXPRESSIONS

12 PERSONAL AUTHOR(S) South, Jesse T.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (*year, month, day*)	15 PAGE COUNT
Master's Thesis	From To	1991, September, 26	100

16 SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES			18 SUBJECT TERMS (*continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUBGROUP	View materialization, Query processing, Semi-materialization, Query modification, Select-project-join expressions.

19 ABSTRACT (*continue on reverse if necessary and identify by block number*)

In conventional relational database systems, a view is a virtual relation whose definition is stored in the systems catalog. When a query is issued on the view, the system retrieves the view from the catalog and modifies the query to an equivalent one on the base relations. Recently several approaches have been proposed that store some form of the computed view as a method for improving the performance of queries on relational databases. This thesis develops a computer program to empirically compare and evaluate three view materialization strategies: query modification, semi-materialization and full materialization. The Program simulates user updates and queries, and measures the cost performance of the three materialization strategies. The strategies are compared for select-project-join expressions under three different view models. The results show that the most efficient view strategy is heavily application dependent. The performance of semi-materialization and full materialization, however, are comparable for most conditions tested, and preferred over the conventional query modification method.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS	Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Magdi N. Kamel	22b TELEPHONE (*Include Area code*) (408) 646-2494	22c OFFICE SYMBOL AS/Ka

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

Approved for public release; distribution is unlimited.

A Performance Analysis of View
Materialization Strategies for
Select-Project-Join Expressions

by

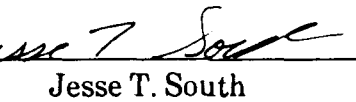
Jesse T. South
Lieutenant, United States Navy
B.S., University of Arizona , 1984

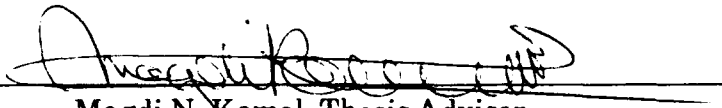
Submitted in partial fulfillment
of the requirements for the degree of

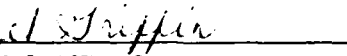
MASTER OF SCIENCE IN INFORMATION SYSTEMS

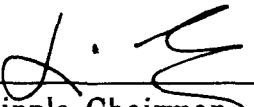
from the

NAVAL POSTGRADUATE SCHOOL
September, 1991

Author: 
Jesse T. South

Approved by: 
Magdi N. Kamel, Thesis Advisor


Rachel Griffin, Second Reader


David R. Whipple, Chairman
Department of Administrative Sciences

ABSTRACT

In conventional relational database systems, a view is a virtual relation whose definition is stored in the systems catalog. When a query is issued on the view, the system retrieves the view from the catalog and modifies the query to an equivalent one on the base relations. Recently several approaches have been proposed that store some form of the computed view as a method for improving the performance of queries on relational databases. This thesis develops a computer program to empirically compare and evaluate three view materialization strategies: query modification, semi-materialization and full materialization. The program simulates user updates and queries, and measures the cost performance of the three materialization strategies. The strategies are compared for select-project-join expressions under three different view models. The results show that the most efficient view strategy is heavily application dependent. The performance of semi-materialization and full materialization, however, are comparable for most conditions tested, and preferred over the conventional query modification method.

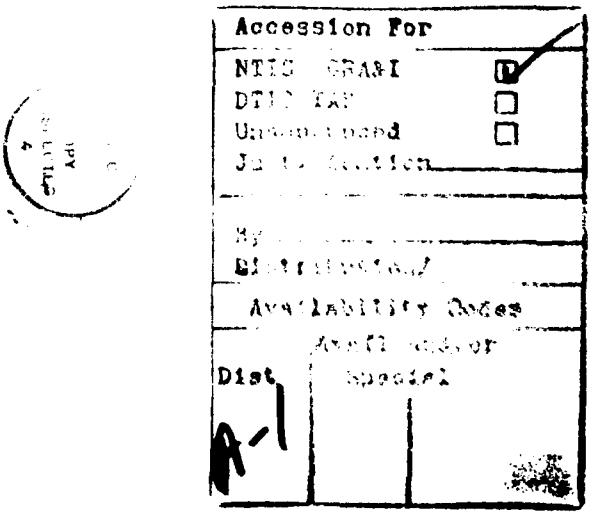


TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

In relational database systems views are commonly used to simplify the conceptual model of the database. A view is defined by a relational expression or query over one or more base relations. In conventional relational database systems, a view is a virtual relation whose definition is stored in the system catalog. When a query is issued on the view, the system retrieves the view definition from the catalog and modifies it to an equivalent one on the base relations. This process is referred to as query modification.

Recently, several proposals have considered storing some form of the materialized view to eliminate the need to reevaluate the view definition every time it is referenced. Two such proposals are full materialization and semi-materialization. In full materialization a copy of the retrieved view is stored in the database. In semi-materialization, however, redundant subsets of the base relations are stored in the database as an intermediate state of computing the view. When a query is issued on the view the subsets are easily joined to form the view.

B. OBJECTIVE

This thesis compares and evaluates the performance of three view materialization strategies: query modification, semi-materialization and full materialization. The objective of this research is to verify empirically the results obtained analytically under different parameter settings. Specifically, the research develops a program that simulates user updates and queries on test databases under varying parameter settings. Cost performance of each materialization strategy is measured, and the results collected and analyzed. The overall objective of the study is to identify the parameters under which each strategy performs the best for different select-project-join expressions.

C. SCOPE OF THE THESIS

This thesis:

1. Develops a program that simulates users updates and queries, measures the cost performance of the materialization strategies, and collects the results.
2. Runs the simulation on different test databases under different parameter settings for select-project-join expressions.
3. Analyzes the results obtained, compares them with analytical results, and draws conclusions.

D. METHODOLOGY

The methodology used for this thesis involves developing a program that simulates user updates and queries, measures

the cost performance of each materialization strategy, and collects the results. Experimental parameters are varied to allow each strategy to be compared under different conditions. Final results are analyzed, compared with analytical results, and conclusions drawn.

The program is written in Microsoft C with embedded SQL commands. The databases and views are managed by the database management system, INGRES (version 8). The simulations are run on a stand-alone 286 personal computer to avoid problems associated with multi-user environments and to guarantee accurate cost measurements.

E. BENEFITS OF STUDY

The performance of processing view strategies directly relates to the performance of real-time applications, such as, the use of surveillance systems for military applications. These systems utilize timed updates of environmental information that is periodically relayed from sensors. These updates essentially trigger a view which must be analyzed and responded to if a hostile presence is detected. As a delayed response could hinder the performance of the system, optimal processing of the view strategies should be performed.

F. ORGANIZATION OF THE STUDY

This thesis is organized as follows. Chapter II discusses views in relational database systems, and presents three different strategies for evaluating views. Chapter III describes the design and implementation of a computer program used to measure the cost performance of the view materialization strategies. Chapter VI analyzes the results of implementing the simulation program to determine the conditions under which each strategy performs the best. Chapter V discusses the conclusions of the study and indicates directions for further research.

II. VIEW MATERIALIZATION STRATEGIES

Views are often used to simplify the conceptual model of a relational database system. Views are defined by relational expressions (queries) over one or more underlying base tables. Views allow users to access and manipulate database tables in a simplified manner.

Views may be virtual or materialized tables. The traditional concept of a view is that of a "virtual" table, i.e., a table that does not physically exist in its own right. However, to the user it appears like a real table. The view definition is stored in the systems catalog (i.e., database dictionary). This definition is retrieved and combined with the query on the view to evaluate the results. A relatively recent approach is that of materialized views. Under this approach the results of evaluating all or part of a view definition may actually be stored. This concept reduces the need to constantly re-evaluate the view definition every time the view is referenced. Three types of approaches for processing queries on views have been proposed. These include query modification, full materialization, and semi-materialization. These strategies are detailed in the following sections.

A. QUERY MODIFICATION

The conventional method for processing queries on views is query modification (Stonebreaker, 1975). Under this approach, a view definition is stored in the systems catalog. When a query is issued on the view, the view definition is retrieved from the catalog and appended to the query. The query is then optimized and executed. An efficient access path is normally chosen, as a good query optimizer should be able to select the best access path for executing the query. Consider the following database schema:

EMP(E#,ENAME,ADDRESS,SALARY,TITLE)

POS(E#,S#,LEVEL,KEYNO)

and the corresponding view definition:

$$GOODEMPS:\ \pi_{e.ENUM,e.ENAME,e.SALARY}(\sigma_{p.LEVEL>10}(EMP \bowtie POS))$$

Now suppose the following query on GOODEMPS is made:

$$\pi_{g.ENUM,g.ENAME}(\sigma_{g.SALARY>50,000}(GOODEMPS))$$

The system converts the initial query into an equivalent query on the underlying base relation:

$$\pi_{e.ENUM,e.ENAME}(\sigma_{e.SALARY>50,000 \wedge p.LEVEL>10}(EMP \bowtie POS))$$

This query is then optimized and processed by the query processor.

B. FULL MATERIALIZATION AND DIFFERENTIAL UPDATES

A materialized view is a stored copy of the result of retrieving the view from the database. Any updates or deletions made to the base relations need to be reflected in the materialized view. The advantage of this method is an increased response time when a query is performed, as much of the work of processing the view has previously been performed. The main disadvantage of this method is the incremental cost of maintaining the materialized view as a result of updates to the base tables[1].

Several algorithms have been proposed to maintain these materialized views. These methods include immediate maintenance (Blakeley, 1986), deferred maintenance (Hanson, 1987), and periodic database snapshots (Adiba,1980)(Lindsay, 1986). Immediate maintenance allows the materialized view to be updated as soon as the new records are inserted into the base tables. This technique allows a good query response time to be achieved. Deferred maintenance, on the other hand, waits as long as possible to update the materialized view. Updates are stored in a temporary file, and the view is not updated until a query is issued on the materialized view. This strategy incurs less overall cost, because several updates may be placed into the temporary file before a query is issued on the materialized view. Finally, updates in

---

[1] Cost here refers to the additional CPU time required to perform the given operation.

periodic database snapshots have no pre-determined periodicity, but may be optionally generated by the user. However, this method does not guarantee an updated materialized view when a query is issued. (Srivastava, 1988)

Certain view expressions could make use of differential update techniques. Instead of recomputing the updated view expression each time the base tables are updated, differential update algorithms identify which tuples must be inserted into or deleted from the current materialized view. With select-project-join expressions, the distributive properties of selection and projection over union, is capitalized on to provide an acceptable differential update algorithm.

Consider the following expression of the materialized view GOODEMPS.

$$GOODEMPS: \pi_{e.ENUM, e.ENAME, e.SALARY}(\sigma_{p.LEVEL>10}(EMP \bowtie POS))$$

Suppose that the materialized view is updated by the insertion of tuples A_1 and deletion of tuples D_1 from the base relation POS. Only the net changes made by these sets of tuples will be included in this transaction. If another different tuple is inserted and then deleted within the same transaction, it will not be represented in the sets A_1 and D_1. The new state of the above expression becomes:

$$GOODEMPS' = \pi_{e.ENUM, e.ENAME, e.SAL}(\sigma_{p.LEVEL>10}((((EMP - D_1) \cup A_1) \bowtie POS))$$

Expanding the above expression, and distributing selection and projection over union, the following is obtained:

$$GOODEMPS' = \pi_{e.ENUM, e.ENAME, e.SAL}(\sigma_{p.LEVEL>10}((EMP \bowtie POS) - (D_1 \bowtie EMP) \cup (A_1 \bowtie EMP)))$$

$$= \pi_{e.ENUM, e.ENAME, e.SALARY}(\sigma_{p.LEVEL>10}(EMP \bowtie POS)) - \pi_{e.ENUM, e.ENAME, e.SALARY}(\sigma_{p.LEVEL>10}(D_1 \bowtie EMP)) \cup \pi_{e.ENUM, e.ENAME, e.SALARY}(\sigma_{e.LEVEL>10}(A_1 \bowtie EMP))$$

$$= GOODEMPS - \pi_{e.ENUM, e.ENAME, e.SALARY}(\sigma_{p.LEVEL>10}(D_1 \bowtie EMP)) \cup \pi_{e.ENUM, e.ENAME, e.SALARY}(\sigma_{p.LEVEL>10}(A_1 \bowtie EMP))$$

Therefore, the materialized view may be updated by computing the last two expressions and then inserting or deleting them from the relation, GOODEMPS. For simplicity, only updates to the base relation, POS, have been considered in this study[2].

The algorithm becomes more complicated as insertions and deletions occur. One reason is that more than one source may have contributed to the tuples in the materialized view. If a deletion transaction occurs and the view is stored with the duplicates removed, it is impossible to know if a record should be removed from the materialized view. To overcome this complication, each fully materialized view should store a duplicate count to indicate the number contributing to each

[2]For a more complete discussion of differential updates see (Blakeley, 1986) and (Kamel, 1991).

tuple in the view when EMP and POS are joined. The count may be either incremented or decremented accordingly each time a insertion or deletion occurs. The tuple may be removed from the view when the count becomes zero during a deletion.(Hanson, 1987)

C. SEMI-MATERIALIZATION

Semi-materialization stores redundant subsets of carefully chosen data from individual base relations (Kamel, 1990). These subsets represent an intermediate state of the view. Each subset is a projection of each base relation of the attribute(s) specified in the view expression and clustered on the join attribute(s). This technique allows for efficient view evaluation and easy maintenance of redundant relations. Updates to the base relations are screened to see if they affect the semi-materialized relations. If this occurs, the updates may be inserted into the redundant subset without the added cost of joining the updates with another relation.

Using the semi-materialization technique, the following redundant relations (clustered on the join attribute ENUM) would be stored for the previously defined view GOODEMPS:

$$EMP' = \pi_{e.ENUM, e.ENAME, e.SALARY}(EMP)$$

$$POS' = \pi_{p.ENUM}(\sigma_{p.LEVEL>10}(POS))$$

When a query is issued on the view, the system converts it into the following query on the redundant relations:

$$GOODEMPS = \pi_{e'.ENUM, e'.ENAME}(\sigma_{e'.SALARY>50,000}(POS' \bowtie EMP'))$$

This process is similar to that illustrated for query modification[3].

Like full materialization, more than one source may have contributed to the tuples in the semi-materialized view. Again, if a deletion transaction occurs and the view is stored with duplicates removed, it is impossible to know if a record should be removed from the semi-materialized view. To overcome this complication, each semi-materialized view should store a duplicate count, indicating the number of tuples in the base relation that contribute to each tuple in the view. Then each time an insertion or deletion occurs, the count may be either incremented or decremented accordingly. The tuple may be removed from the semi-materialized view when the count becomes zero during a deletions.(Hanson, 1987)

[3]This technique is fully discussed in (Kamel, 1991).

III. PROGRAM DESIGN

This chapter presents the design and implementation of a computer program used to measure the cost performance of the three materialization strategies. The main function of the program is to simulate user updates and queries and to measure and report the performance of each materialization strategy. The simulation program is written in C with embedded SQL commands to access the INGRES relational database.

A. OPERATING ENVIRONMENT

The database consists of two base relations with the profiles specified in Table 3.1 and 3.2. In these tables, VAL is the number of unique values of each attribute, SIZE is the size of each attribute, and CARD is the cardinality of the relation. The important parameters of the analysis are described in Table 3.3. Updates to the base relations and queries on the view are the only two database operations the program may perform. It is assumed there will be k update transactions, each modifying l tuples, and q queries on the view. To prevent the query from benefiting from the indexing used for the view, the view predicate is different from the query predicate. The indexing structure used is a compressed

B-tree[4]. Performance is measured as the average elapsed time per query over all k updates and all q queries.

B. GENERAL PROGRAM DESCRIPTION

The purpose of this program is to simulate user updates and queries to measure the performance of the three materialization strategies. The performance is measured by varying each one of the following parameters, while keeping the other parameters constant.

1. The total fraction of updates to the number of operation, P. This may be controlled by varying the value of parameters k and q.
2. The selectivity of the view, f_v. This is controlled by varying the value of the parameter used in the view predicate.
3. The selectivity of the query, f_q. This is controlled by varying the value of the parameter used in the query predicate.
4. The cardinality (i.e., the number of tuples) used in the update transaction, l. This may be controlled by varying the size of the update tuples generated by the data generation program.
5. The cardinality of the base relations, N. This may be controlled by varying the size of the POS and EMP tables prior to the execution of the program.

The above parameters are considered to be the most sensitive in determining the performance of each view strategy (Hanson, 1987)(Kamel, 1990). The cost of updates and queries

---

[4]PC INGRES limits the storage structure to compressed heap and compressed B-tree.

is determined by computing the total elapsed time per query to update the materialized view or the redundant relations and to perform the queries on the view.

The principal modules of the program are the control module (The Main Module) and the view materialization modules. Figure 3.1 shows a simplified schematic diagram of the simulation program. This diagram illustrates the relationship between the principal modules and each functional phase. The complete program design is discussed in the next section. The control module oversees the activities of the entire program. Input from a control file directs the control module as to which operations to perform. The first operation initialize the test database. The view strategies are then tested, and the results are written to two output files.

Each of the view materialization strategies has its own functional module. Each module measures the elapsed time for performing either an update transaction or a query request.

C. MAIN MODULE DESCRIPTION

This section discusses the principal modules of the simulation program in detail. The structured charts demonstrating the data flows and system hierarchy are included in Appendix A. The program source code is listed in Appendix B.

1. Control Module (Main Module)

The purpose of the control module is to direct the activity of the entire program. This includes controlling inputs, invoking the view modules, and overseeing output results. The module reads data from two control files. The first file, DBINFO.DAT, shown in Table 3.4 contains information about the database (i.e., cardinality, characteristics of view and query predicates). The second file, CNTRL.DAT, listed in Table 3.5 contains the parameters used for the run. Output from the program is routed to the Write Final Result module, where the summary results (Table 3.6) are written to a text file.

Functionally, the module reads data from the control file. For each record of the control file, the module first calls the initialization module, passing the value of the view predicate. The view strategy modules are then invoked to perform the updates and queries. The values of the parameters k and q determine how many updates and queries are performed respectively. Finally, the control module invokes the modules that compute the average cost per query, table counts, and predicate selectivities.

2. Initialize Test Database Module

The purpose of this module is to initialize the test database. Each of the subordinate modules called by the initialization module use embedded SQL commands to create the

database tables, views and indexes, and to copy the table contents from text files. The module accepts as input the view predicate from the main module.

Functionally, the module makes a call to the database management system to destroy the old test database and then create a new one. The module then invokes the following modules to create the initial setup of the database.

1. The Create Tables module creates all the necessary base, materialized view, and redundant tables.
2. The Copy Tables module loads the base tables from the *POSDAT* and *EMPDAT* text files using the SQL copy command.
3. The Copy Semi- and Full Materialization module accepts the view predicate as input using the SQL insert command. Data from the base tables that meet the view definition are inserted into the materialized view and redundant tables.
4. The Create Table Index module modifies the table storage structures to compressed B-tree with the SQL modify command.

3. Query Modification Module

The purpose of the query modification module is to perform an update transaction or query request, and to measure the response time for queries. The module accepts as inputs the type of operation to be performed (i.e., an update or query), the value of the view and query predicates, the running elapsed query time for query modification, and the name of the detailed output file. The module returns the current running elapsed time for queries.

Functionally, a control character determines whether an update or query is to be performed. If an update is selected, the module inserts the update records into base relation, R1 (POS). If a query is selected, the module performs a query on the view and measures its performance. The module then computes the new running elapsed time and writes the elapsed time for that run to the detailed output file.

4. Full Materialization Module

The purpose of the full materialization module is to perform an update transaction or query request and to measure their performance. The module accepts as input the type of operation to be performed (i.e., update or query), the value of the view and query predicates, the running elapsed query time for full materialization thus far, and the name of the detailed output file. The module returns the current running elapsed time.

Functionally, a control character determines whether an update or query is to be performed. If an update is selected, the module inserts the update records into the base table, R1 (POS). It then inserts those records that meet the view definition into the fully materialized view FULLMAT and measures its performance. If a query is selected, the module performs a query on the view and measures its performance. The module then computes the new running elapsed time and

writes the elapsed time for that run to the detailed output file.

5. Semi Materialization Module

The purpose of the semi-materialization module is to perform an update transaction or query request, and to time their performance. The module accepts as input the type of operation to be performed (i.e., update or query), the value of the view and query predicates, the running elapsed query time for semi-materialization, and the name of the detailed output file. The module returns the current running elapsed time.

Functionally, the control character determines whether an update or query is to be performed. If an update is selected, the module inserts the update records into the base table, R1 (POS). It then inserts those records that meet the definition of the semi-materialized relations into R1' (POS_PRIM), and measures its performance. If a query is selected, the module performs a query on the view and measures its performance. The module then computes the new running elapsed time and writes the elapsed time for that run to the detailed output file.

6. Compute Average Time Module

The purpose of this module is to compute the average total cost per query for each view materialization strategy. The module accepts as input the number of queries performed

and the total elapsed time to perform all queries on the view and updates to the materialized view or redundant relations. The average cost per query for each strategy is returned.

7. Compute f_v, f_q and P Module

The purpose of this module is to compute the ratio values of the selectivity of the view, f_v, the selectivity of the query, f_q, and the probability of an update, P. The module takes as input the base value, the increment used, the number of values in the predicate range (VAL) and predicate values for the view and query, and the number of updates, k, and queries, q. The module returns the estimated view selectivity, the estimated query selectivity and the update probability.

The variables base, increment and VAL are used to define the range of values that the view and query predicate may take. For example, the query predicate *"salary"* has a range from 5,000 to 50,000. This range may be determined using the values of the variables base, increment and VAL. For the query predicate *"salary"*, their values are 5,000, 5,000, and ten, respectively. The base starts at 5000 and is incremented by 5,000 ten times to obtain the desired range. The module also determines the probability of an update by dividing the number of updates, k, by the total number of updates and queries, $k+q$, (i.e., $k/(k+q)$).

8. Compute Table Counts Module

The purpose of this module is to calculate the number of records in the base, view, and query. The base is the count of all the tuples that could conceivably be derived by joining of relations R1 (POS) and R2 (EMP). For a one-to-one join condition, the base would be equal to the larger of these two relations. Since there is a one to many relationship between EMP and POS the larger of the two relations is POS (R1). The values of the base, view and query are then used to determine the percentage of records that are actually in the view and in the query. The module accepts the query predicate as input. It returns the count of the base, view, and query, and the actual selectivity of the query and the view.

Functionally, the module creates a temporary table of all possible records that could be in the view by joining the tables, R1 (POS) and R2 (EMP). Joining the relations in this manner permits conditions in which relation R2 is larger than relation R1. It also allows an accurate count of the tuples in the base for one-to-one join conditions. An SQL count is performed on the temporary table to determine the number of records in the base, view, and query. These counts are used by the module to determine an actual percentage of the records in the view and the query.

D. PROGRAM OPERATION

1. Starting the Simulation

The simulation program may be initiated from a DOS batch file or by typing the file name directly at the DOS prompt. The following conditions must exist to operate the program successfully:

1. The files in Table 3.7 must be located in the working directory.
2. The INGRES relational database management system must be installed and put into the DOS directory path.
3. Sufficient disk space, as the simulation is very hard drive intensive. An *N* size of 5,000 records used for this analysis required five megabytes of disk space.
4. Patience. A single simulation run may take three hours or longer to complete.

To run a simulation, the program name is entered from the working directory. Execution of the program from a batch file is slightly more complicated. Table 3.8 illustrates a sample batch file, in which the database text files were created and the control file, containing the parameters for testing the strategies, was copied to the working directory. The program was then executed. The batch file may be set up to run the simulation in several different ways.

2. Execution

The parameters are read into the program from a control file. Table 3.5 identifies the fields of a record of the control file. To investigate the effects of a single

parameter on the cost per query, the value of the parameter is varied over a predetermined range. Figure 3.2 illustrates an example of a control file that could be used to vary the probability of an update, *P*. In this example, the values of the third field, the number of updates *k*, and fourth field, the number of queries *q*, are varied to study the effect of varying the probability of update on the cost per query.

The program itself may be broken down into three phases: an initialization phase, a testing phase, and a computation phase. The following sections describe these three phases in more detail.

a. Initialization Phase

In the initialization phase the database is initialized and populated. The old test database is destroyed and a new one created. This includes the creation of all relations (base and redundant), views, and indexes required by the simulation. Separate relations are built for each materialization strategy, so that each strategy may operate under similar environments[5]. Data is copied into the database relations from text files created by a separate data generation program. The last function of the initialization phase modifies the database relations to a compressed B-tree

[5]Three relations are actually created for each strategy to allow for testing of three way joins in future testing.

structure on the keys specified, and allows for the creation of secondary indexes[6].

b. Testing Phase

In the testing phase, the program measures the total elapsed time to update the redundant relations and to query the views for each strategy. Two subloops exist in this phase in which each of the strategies is tested according to the parameters set in the control file. The values q and k determine the number of times each loop will run. Each strategy is then tested by its functional module.

The view materialization modules operate in essentially the same manner. Each contain a case statement that either updates the relations used by each strategy or performs a query on the view. Computing the elapsed time is the last function of the three modules. The operation being timed is the main difference between the modules. The performance of the queries is measured the same for all three methods, but the updates are measured only for semi-materialization and full materialization. Time is measured for updates to the redundant relations, and as query modification has no redundant relations no time measurement is required.

---

[6]To change modify and index keys the source code needs to be changed and the program recompiled.

c. Computation and Report Phase

The final phase of the program consists of four functions. First, the average cost per query for each strategy is determined. This is accomplished by averaging the total elapsed time for each strategy over the number of queries, q, performed during that run. Second, the estimated values for the selectivity of the view, (f_v), selectivity of the query, (f_q), and the probability of an update are calculated from input provided by the control files. This process verifies that the actual parameter selectivities match the estimated ones. The third function determines the size of the base, view, and query. Base is defined here as the largest possible relation that could conceivably be derived by joining the relations, EMP and POS. For a one-to-one join between R1 (POS) and R2 (EMP), base is the size of relation R1. A count of the tuples in the fully materialized view is used to determine the size of the view and the size of the query. These three values are then used to calculate the actual values of f_v (f_{va}) and f_q (f_{qa}). The final step writes the results in the final result file. The results are then appended to the previous results, allowing the program to operate repeatedly within a batch file. The program continues until no further entries exist in the control file. The program then reinitializes the database, tests the strategies, and computes the results for each run.

TABLE 3.1 PROFILE OF RELATION R1 (POS)

	E#	S#	LEVEL	ACCREDITINFO
VAL	500	200	10	5,000
SIZE	6	6	2	86

CARD(POS) = 5000

TABLE 3.2 PROFILE OF RELATION R2 (EMP)

	E#	D#	ENAME	ADDRESS	SALARY	TITLE	JOBDESC
VAL	500	10	500	500	10	10	10
SIZE	6	6	20	70	8	30	60

CARD(EMP) = 500

TABLE 3.3 PARAMETERS IMPORTANT TO ANALYSIS

Parameter	Definition
N	Cardinality of relation R1
k	Number of update transaction on base relations
l	Total number of tuples modified by each update transaction
q	Number of times the view is queried
P	Probability that a given operation is an update ($P = k/(k+q)$)
f_v	Selectivity of view predicate (fraction of tuples in the view)
f_q	Selectivity of query predicate (fraction of tuples retrieved by the query on the view)
f_{R2}	Size of relation R2 as fraction of R1

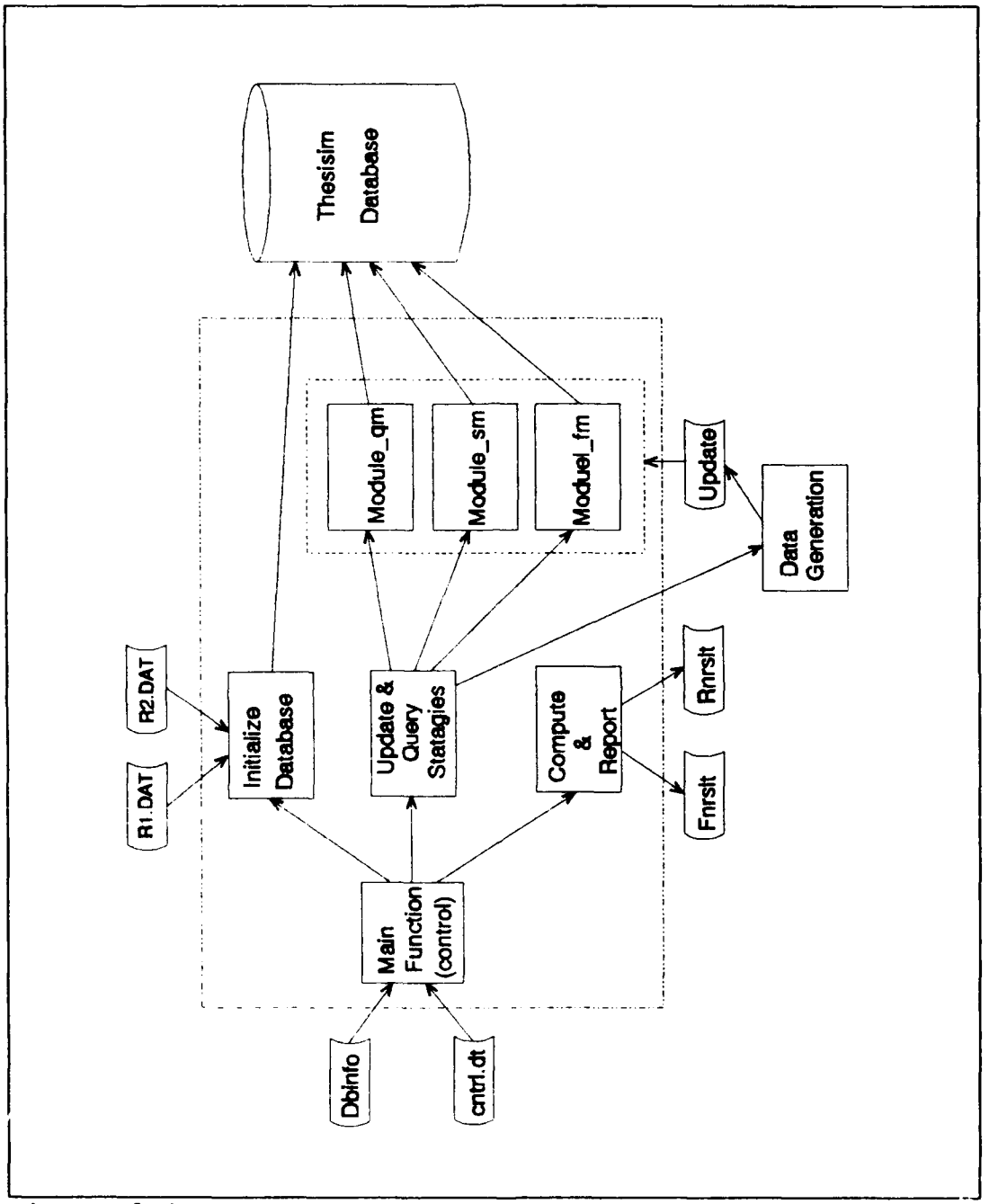


Figure 3.1 Schematic Diagram of Simulation Program

TABLE 3.4 CONTENTS OF DBINFO FILE

Variable	Description
ecard	Cardinality of the employee text file
pcard	Cardinality of the possess text file
scard	Cardinality of the skill text file (not used in this analysis)
vmax	Number of view predicates, used to determine the range
vbase	Lower bound of the range of view predicate
vincr	This value is used to increment the view predicate range
qmax	Number of query predicates, used to determine the range
qbase	Lower bound of the range of query predicate
qincr	This value is used to increment the query predicate range

TABLE 3.5 CONTENTS OF CONTROL FILE

parameter	description
viewcut	f_v, the selectivity of the view
querycut	f_q, the selectivity of the query
k	Number of updates performed
q	Number of queries performed
updat_siz	l, number of tuples in the update
parameter	Holds the name of parameter being updated
update_rel	Holds the name of relation being updated

```
10 50000 1 9 25 prob possess
10 50000 2 8 25 prob possess
10 50000 3 7 25 prob possess
10 50000 4 6 25 prob possess
10 50000 5 5 25 prob possess
10 50000 6 4 25 prob possess
10 50000 3 7 25 prob possess
10 50000 2 8 25 prob possess
10 50000 1 9 25 prob possess
```

Figure 3.2 Sample Control File.

TABLE 3.6 CONTENTS OF SUMMERY RESULTS

Variable	Description
RUN#	The number of control file entries
VCUT	View predicate value
QCUT	Query predicate value
#TUP	Value of *1*, number in the update
BASE	Number of records in the base
VIEW	Number of records in the view
QUERY	Number of records in the query
FV	Selectivity of the view
FVA	Actual selectivity of the view
FQ	Selectivity of the query
FQA	Actual selectivity of the query
P	Probability of an update
TIMEQM	Average elapsed time of a query for query modification
TIMESM	Average elapsed time of a query for semi-materialization
TIMEFM	Average elapsed time of a query for full materialization

TABLE 3.7 REQUIRED PROGRAM FILES

Filename	Description
SIMVIEW.EXE	The executable simulation program.
DATAGEN.EXE	The data generation program, used to produce all data text files.
CNTRL.DAT	This file contains the control information for the simulation.
DBINFO.DAT	This file contains information pertinent to the data text files, i.e., cardinality, view and query predicate range information.
DATA_IN	This is the control file for the data generation program.
	(The following files are created by the data generation program)
POSDAT.DAT	This text file contains the data for the POS relation.
EMPDAT.DAT	This text file contains the data for the EMP relation.
SKILDAT.DAT	This text file contains the data for the SKILL relation, (not currently used in this simulation).
UPDATE.DAT	This text file is created during execution of the program, and is used to update the base relations.
	(The following files are created by the simulation program)
FNLRSLT.DAT	This file contains the summary results of the simulation.
RNRSLT.DAT	This file contains the detailed results of the simulation.

TABLE 3.8 SAMPLE EXECUTABLE BATCH FILE.

```
datagen.exe
copy cntrlp.dat cntrl.dat
simview.exe
```

IV. PERFORMANCE ANALYSIS

The performance characteristics of each materialization strategy are different. In this chapter, the results of implementing the simulation program are analyzed and the condition under which each strategy performs the best are determined.

The view simulation is carried out on a 286 personal computer running the DOS operating system. The program is written in C with embedded SQL commands that access the relational database system INGRES.

A. EXPERIMENTAL SETUP

The database consists of two base tables. The profiles of the base tables were shown in Table 3.1 and 3.2, respectively. The parameters important to the experiment were shown in Table 3.3 of Chapter III. The default values of these parameters, unless otherwise stated, are presented in Table 4.1. The parameters that are used in the simulation are:

1. The fraction of updates to the total number of operations, P. This is controlled by varying k and q.
2. The selectivity of the view predicate, f_v. This is controlled by varying the value of viewcut in the view definition.
3. The selectivity of the query predicate, f_q. This is controlled by varying the value of querycut in the query definition.

4. The number of tuples modified by each update, l. This is controlled by varying the size of the update relation.

5. The cardinality of the base relation R1, N. This is controlled by varying the size of the text files used to build the base relations, POSDAT and EMPDAT.

Each model is tested by varying one parameter at a time over a suitable range, while keeping the other parameters at their default values. Two types of operations are performed: k update transactions and q queries on the view. Performance of each strategy is measured by the average elapsed time per query, over all updates and queries.

B. PERFORMANCE TESTING

In the following sections, the performance of each of the three techniques is analyzed for select-project-join view expressions of the following form:

$$V = \pi_{R1.fields,R2.fields}(\sigma_{C(R1)}(R1 \bowtie R2))$$

where R1 contains N tuples and R2 has $f_{R2} \cdot N$ tuples. The selection predicate, C(R1), restricts relation, R1, with selectivity, f_v. Three different models are considered:

1. Model-1 - A select-project-join expression of the above type, such that every tuple in R1 that satisfies the selection predicate, C, joins with exactly one tuple in R2. All updates are applied to R1, and R2 is never updated.

2. Model-2 - An expression of the above type, where the requirement that every tuple from relation R1 join with exactly one tuple in R2 is relaxed.

3. Model-3 - An expression of the above type, where updates are applied to both relations R1 and R2.

The results from each model are plotted to show the differences between each strategy. Each graph plots the total cost per query for each strategy versus the parameter under investigation.

1. Model-1

In Model-1, the performance of the three strategies is analyzed for the above select-project-join expression while varying the access paths of the base relations. The view definition and query used are given in Table 4.2. The access paths used for the three program simulations are shown in Tables 4.3, 4.4 and 4.5. In the first simulation, Model-1a, the access paths are set to be the most efficient for semi-materialization and full materialization. The second simulation, Model-1b, sets the access path to the join attribute, *ENUM*, for all relations including the fully materialized view. The third simulation, Model-1c, adds a secondary index to the access path setup of the second simulation.

Each model is executed by varying each of the five parameters listed in the previous section. The differences in cost per query for query modification, full materialization, and semi-materialization are plotted against each parameter.

a. Results for Model-1a

Figure 4.1 displays the cost of a query (in seconds) against the probability that an operation is an update, *P*. Except for high values of *P*, both full materialization and semi-materialization are superior to query modification. At values of *P* from 0 to .4, full materialization performs slightly better than semi-materialization. Semi-materialization exhibits a slightly higher cost to perform the query. The main advantage of full materialization is its lower cost of performing the query. At *P* values greater than .4, the cost of full materialization rises considerably as the cost of maintaining the materialized view overwhelms the small query cost. The semi-materialization performance is relatively stable for all values of *P* less than .7 because of lower maintenance cost. At values greater than .7, the semi-materialization maintenance costs begin to rise significantly. The semi-materialization performs better than query modification for values of *P* less than .92. The cost of query modification is relatively level for all values of *P*. It performs better than full materialization for values of *P* greater than .8, and better than semi-materialization for values of *P* greater than .92.

Figure 4.2 plots the cost-per-query versus the selectivity of the view, f_v. Semi-materialization is the preferred method for all values of f_v. At f_v values less than

.2, the cost for full materialization and semi-materialization are comparable as maintenance cost for full materialization tend to be low. However, as f_v increases, full materialization maintenance costs are higher than that for semi-materialization.

The cost-per-query versus the selectivity of the query, f_q, is presented in Figure 4.3. Semi-materialization and full materialization again perform better than query modification for all values of f_q. However in this situation, full materialization performs better than semi-materialization over almost the entire range of f_q. The reason that full materialization is the preferred method is that the cost of scanning the semi-materialized relations increases as the fraction of the query increases.

Figure 4.4 graphs the cost-per-query to the number of tuples in the update, l. Semi-materialization is the preferred strategy for values of l greater than 14. The overhead of maintaining the fully materialized view outweighs the small cost of performing the query on the view.

Figure 4.5 shows the performance of the three strategies versus the cardinality of the base relation R1. Both semi-materialization and full materialization performance is better than query modification. Semi-materialization is favored at lower values of N (less than 7500) and full materialization is favored at higher values of N. As the size of the base relation, N, increased, the cost of scanning the

redundant relations to construct the view reduced the performance of semi-materialization.

b. Results for Model-1b

The pattern of the results for Model-1b were very similar to the results of Model-1a. The small differences in the graphs were due to changes in the amount of time the query processor required to perform the query. As the access paths were not as efficient as in Model-1a, the query processor required more time to scan the database relations.

Figure 4.6 shows the cost-per-query versus the probability of an update. Semi-materialization and full materialization performed better than query modification for all values of P tested. However, at values of P greater than .9, a trend emerged that indicated that query modification would perform best. As with Model-1a, full materialization is slightly better than semi-materialization for values of P less than .4. Conversely, semi-materialization is better for P values greater than .4.

For the selectivity of the view, as shown in Figure 4.7, semi-materialization and full materialization perform better than query modification for all values of f_v. Semi-materialization is also the preferred method for almost the entire range of f_v. Similar to Model-1a, as f_v increases the cost of maintaining the full materialization increases the cost of this strategy over semi-materialization.

Full materialization is the favored method for values of the selectivity of the query, f_q greater than .15 as shown in Figure 4.8. In this instance, the cost of scanning the redundant relation for semi-materialization increases as the fraction of tuples retrieved in the view increases.

For the number of tuples modified by an update, Figure 4.9, semi-materialization is shown as the preferred method for values of l greater than 10. Again, this is due to full materialization higher maintenance costs as the number of tuples per update increases.

For the cardinality of the base relation versus the cost-per-query, as indicated in Figure 4.10, full materialization is the preferred method for values of N greater than 7,500. Again, as the size of the base relations increased the cost of scanning the redundant relations increased. The scanning cost of query modification rise dramatically in this model.

c. Results for Model-1c

In Model-1c, the addition of a secondary index showed an improved performance of the view materialization strategies. The basic trends, however, remained the same. Semi-materialization and full materialization perform better than query modification for most parameter settings.

Figure 4.11 gives the cost-per-query versus the probability of an update. Semi-materialization and full

materialization once again perform better than query modification for most values of *P*. Query modification cost per query dropped by more than half, as the secondary index allowed for a more efficient access path for scanning the relations to construct the view and perform the query. Semi-materialization also improved slightly over Model-1b and performed better than full materialization for values of *P* greater than .3. Full materialization improved at lower values of *P*, but degraded at higher values, because the added cost of maintaining the secondary index is greater than the time saved from increased query response.

Model-1c had its most obvious affect on the cost-per-query versus the selectivity of the view (Figure 4.12). Semi-materialization is still the preferred method, but the addition of a secondary index to Model-1b improved full materialization considerably at higher values of f_v. As the selectivity of the view increases, the improvement in response time is greater than the increase in the cost of maintaining the additional index. While this model improves the performance of full materialization, it is not the preferred method. The additional index also improved query modification at lower values of f_v where scanning costs are low.

Figure 4.13 plots the cost-per-query against the selectivity of the query. Semi-materialization performance improved when compared to full materialization. For values less than .4, semi-materialization is the better strategy.

This is because a higher increase in maintenance costs is incurred for full materialization from the addition of the secondary index. At values greater than .4, full materialization performs better because of the higher cost to process the query in semi-materialization.

The cost-per-query versus the number of tuples in the update is graphed in Figure 4.14. Semi-materialization is the preferred method for all values of *l*, even at values less than 10, where full materialization was previously the preferred method. The reason for that is because there is a higher cost to maintain the secondary index for full materialization.

The Cardinality of the base relations against the cost-per-query is shown in Figure 4.15. Semi-materialization performed better than full materialization for values of *N* less than 12,500. The additional maintenance incurred by full materialization to maintain the secondary index overwhelms the increased cost of scanning the redundant relations for semi-materialization. At higher values of *N*, (greater than 12,500) full materialization is the preferred method, as scanning cost begin to rise faster than maintenance cost.

2. Model-2

In Model-2, the view materialization strategies are compared for the same select-project-join expression as above. The strategies are the same as before with the exception that

the condition that all tuples from relation R1 match with exactly one tuple in R2 is relaxed. The view definition and access paths used are the same as for Model-1a (Tables 4.2 and 4.3). To relax the one-to-one join condition, the attribute, *ENUM* (i.e., employee number) of the POS (R1) relation was given a range of values larger than the actual number of employees. In this simulation, the range of employee numbers in relation POS was randomly distributed from one to 1000, while the actual range of the EMP relation (R2) was one to 500. This reduced the one-to-one relationship by about 50 percent, because only half of the tuples in the POS relation have matching tuples in the EMP relation.

a. Results for Model-2

Relaxing the one-to-one join condition reduces the number of tuples in the view and query. This reduced view size affects each view strategy differently. There is virtually no change for query modification because the base tables remain the same. Therefore, the performance of a query incurs virtually the same cost. A slight improvement is shown with semi-materialization, but the overall trends of the graphs and the cost per query are virtually the same as the previous model. A minor improvement occurs in the cost of maintaining the redundant relations. This reduction is thought to occur due to fewer update tuples meeting the view

definition. The cost per query showed a slight improvement, probably because the query is joining fewer records.

The condition in which less than a one-to-one join condition exists appears to be most favorable to full materialization. Compared to Model-1a, the cost-per-query is lower for most parameter settings. The less than one-to-one join condition reduces the size of the view thus allowing for cheaper costs in maintaining and querying the view. While the same cost is incurred for screening the update against the view definition, a lower cost is incurred for maintaining the fully materialized view storage structure. A smaller view is also much faster to query.

Figure 4.16 plots the probability of an operation being an update versus the cost-per-query. Except for high values of *P*, semi-materialization and full materialization are preferable to query modification. Query modification is favored for values of *P* greater than .8 versus full materialization, and .9 versus semi-materialization. Full materialization is preferred for values of *P* less than .45, a slight improvement over Model-1a thought to be caused by the smaller size of the view. Semi-materialization is better than full materialization at values greater than .45, because the maintenance costs involved in full materialization overwhelm any cost savings resulting from the smaller view, or quicker query response.

The selectivity of the view is graphed in Figure 4.17. The performance of semi-materialization and full materialization are surprisingly comparable for all values f_v. The cost-per-query for full materialization dropped considerably from Model-1a, as fewer records meet the view definition. With fewer records in the view, maintenance costs are lower. Semi-materialization performance improves only slightly. As the redundant relation, R1 (POS_PRIM), still contains all tuples that meet the view condition. Tuples are not eliminated from the partially processed view until a query is issued and the join condition is met. As a result, semi-materialization performance stays relatively stable, while full materialization improves considerably. However, even with this improvement, semi-materialization is the preferred strategy for all values of f_v. Query modification is never the preferred strategy.

Figure 4.18 shows the cost-per-query versus the selectivity of the query. In this model, reducing the one-to-one join condition reduces the number of tuples retrieved in the query. As a result semi-materialization performance improves at higher values of f_q. Nevertheless, increasing values of f_q causes the cost of scanning the redundant relations for semi-materialization to increase. This effect makes full materialization the superior method for values of f_q greater than .18. This result is more pronounced in this model, as full materialization maintenance costs are lower.

The number of tuples in the update is graphed in Figure 4.19. Again, semi-materialization and full materialization are the preferred methods over query modification. Both semi-materialization and full materialization are comparable for all values of *l*. At values less than 20, the two strategies are equal. At higher values, semi-materialization is only slightly better than full materialization. The closeness of the two strategies is related to the reduced size of the view. For full materialization, this results in lower maintenance and query costs, while semi-materialization costs remain similar to those in Model-1a.

Figure 4.20 shows the cardinality of the base relations versus the cost-per-query. The results are similar to those found in Figure 4.5. Full materialization is the preferred method for values of *N* greater than 7,500 as the cost of scanning the redundant relation degrades the performance of semi-materialization at higher values of *N*. In this situation increasing the size of the base relations increases the size of the query.

3. Model-3

In this section, the three materialization strategies are compared while applying updates to two base relations. For Model-3, the view definition and access paths will be the same as in Model-1a (Tables 4.2,4.3). The condition that

every tuple in R1, matching the selection predicate, join with exactly one tuple in R1, still applies. To keep the size of the base relations in proportion with each other, updates to R1 contain l tuples and updates to R2 have $f_{R2} \cdot l$ tuples. For this model, the same number of updates are applied to both relations. For example, if R1 is updated twice, then R2 is updated twice. The model tests the strategies for the same parameters and conditions as in Model-1a, except that the additional cost of maintaining R2 is recorded.

a. Results for Model-3

Adding update transactions to both base relations had some interesting results. As expected, query modification was not affected. Since query modification has no redundant relations, it incurs no additional maintenance. The same result was true for semi-materialization in which redundant relations are projections of single base relations. Updates to the redundant relations require simple inserts of the tuples meeting the view definition. The cost of maintenance is dependent on the size of the redundant relation. In this model, semi-materialization is slightly faster than it was in Model-1a as updates are split between two relations. Since R2 is smaller than R1 less maintenance time is required. Conversely, full materialization is adversely affected by updating both base relations. In full materialization, the updates inserted into one base relation must be screened and

joined with tuples from the second base relation before they can be inserted into the fully materialized view. Unlike semi-materialization, updates to R2 experience higher maintenance because they must join with R1 (a larger relation) requiring more time for scanning.

The cost-per-query versus the probability that an operation is an update is given in Figure 4.21. Except for high values of *P*, semi-materialization and full materialization are superior to query modification. Full materialization is preferred for values of *P* less than .2, in which semi-materialization incurs a higher maintenance cost to perform the query. For values greater than .2, semi-materialization is clearly the superior technique. Its performance shows very little change from Model-1a as inserts are performed on a single redundant relation for both update operations. Full materialization, however, requires that updates intended for R2 are joined with records matching the view predicate in R1. The size of R1 makes updates to R2 considerably more costly, due to the increased scanning time. Detailed results show that the cost of maintaining R2 actually triples for full materialization at higher values of *P*. Semi-materialization maintenance costs may actually be less, because less time is required to update the index for the smaller relation, R2'. Query modification was the preferred method for values of *P* greater than .65 versus full materialization, and .9 versus semi-materialization. At these

points, the cost of maintaining the materialized view and redundant relations overwhelms the small cost savings in response time.

Figure 4.22 shows the total cost-per-query versus the selectivity of the view, f_v. Semi-materialization is clearly the superior method for all values of f_v. When comparing the results to Model-1a, additional updates to R2 are shown to cause full materialization to increase at a much steeper rate. The graphs for semi-materialization and query modification are virtually unchanged. This result is not surprising for query modification, as no added maintenance needs to be accounted for. Semi-materialization performs slightly better, than in Model-1a at higher values of f_v. This increase in performance occurs because half the updates in this model are applied to a smaller relation, R2. Smaller relations require less time to maintain their indexes.

Figure 4.23 plots the selectivity of the query against the cost-per-query. Semi-materialization and full materialization are better than query modification for all values of f_q. The additional maintenance incurred by full materialization, however, make semi-materialization the favored method for most values of f_q. It is not until f_q reaches .78 that the cost of scanning the redundant relations make full materialization the preferred strategy.

In Figure 4.24, the number of tuples in the update is graphed. Semi-materialization is the preferred method at

all values of *l*. At lower values of *l*, where full materialization had previously been preferred, increased maintenance costs affect the results. Thus, full materialization is never comparable to semi-materialization in this situation.

The cost-per-query is plotted against the cardinality of the base relations in Figure 4.25. Semi_materialization and full materialization are both preferred to query modification for all value of *N* tested. However, in this instance semi-materialization is also favored over full materialization for all values tested. The additional updates are shown to cause full materialization maintenance costs to increase, because they must be joined with R1, which requires more time to scan. The performance of semi-materialization shows little change from Model-1a, as the inserts are performed on a single relation for the update operations.

TABLE 4.1 PARAMETER DEFAULT VALUES

N	5,000	P	.5
k	10	f_v	.1
l	25	f_q	.1
q	10	f_{R2}	.1

TABLE 4.2 VIEW DEFINITION AND QUERY ON VIEW

	Expression
View:	CREATE VIEW GOODEMPS (ENUM,ENAME,SALARY) SELECT e.ENUM,e.ENAME,e.SALARY FROM e,p WHERE e.ENUM = p.ENUM AND p.LEVEL > view predicate
Query:	SELECT ENUM, ENAME FROM GOODEMPS WHERE SALARY > query predicate

TABLE 4.3 DEFAULT ACCESS PATHS

Base and Redundant Relation(s)	Access path
R1 (POS)	Clustered index on field used in view predicate
R2 (EMP)	Clustered index on join field
R1'(POS')	Clustered index on join field
R2'(EMP')	Clustered index on field used in query predicate
MATVIEW	Clustered index on field used in query predicate

TABLE 4.4 ACCESS PATHS FOR MODEL-1B

Base and Redundant Relation(s)	Access path
R1 (POS)	Clustered index on join field
R2 (EMP)	Clustered index on join field
R1'(POS')	Clustered index on join field
R2'(EMP')	Clustered index on join field
MATVIEW	Clustered index on join field

TABLE 4.5 ACCESS PATHS FOR MODEL-1C

Base and Redundant Relation(s)	Access path
R1 (POS)	Clustered index on join field, with secondary index on view predicate
R2 (EMP)	Clustered index on join field, with secondary index on query predicate
R1'(POS')	Clustered index on join field
R2'(EMP')	Clustered index on join field, with secondary index on query predicate
MATVIEW	Clustered index on join field, with secondary index on query predicate

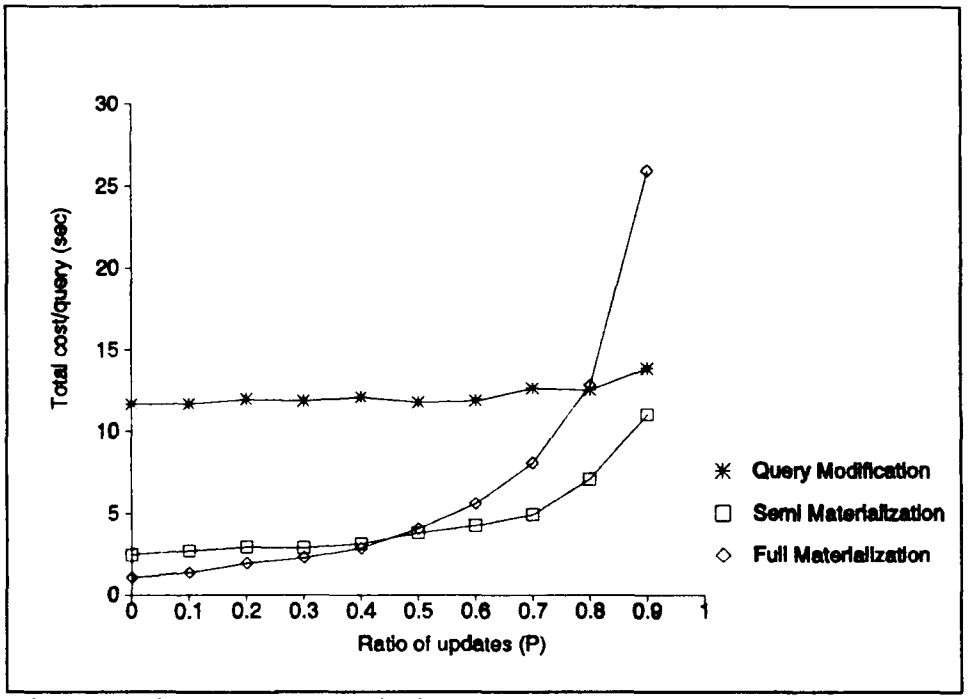


Figure 4.1 Probability of an Update versus the Cost-Per-Query for Model-1a.

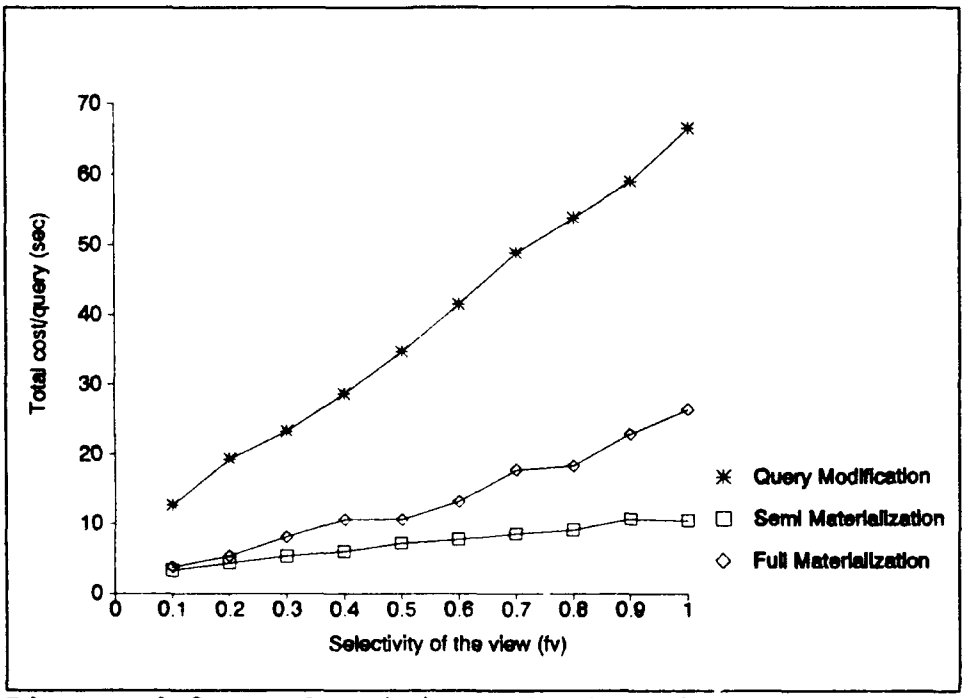


Figure 4.2 Selectivity of the View versus the Cost-Per-Query for Model-1a.

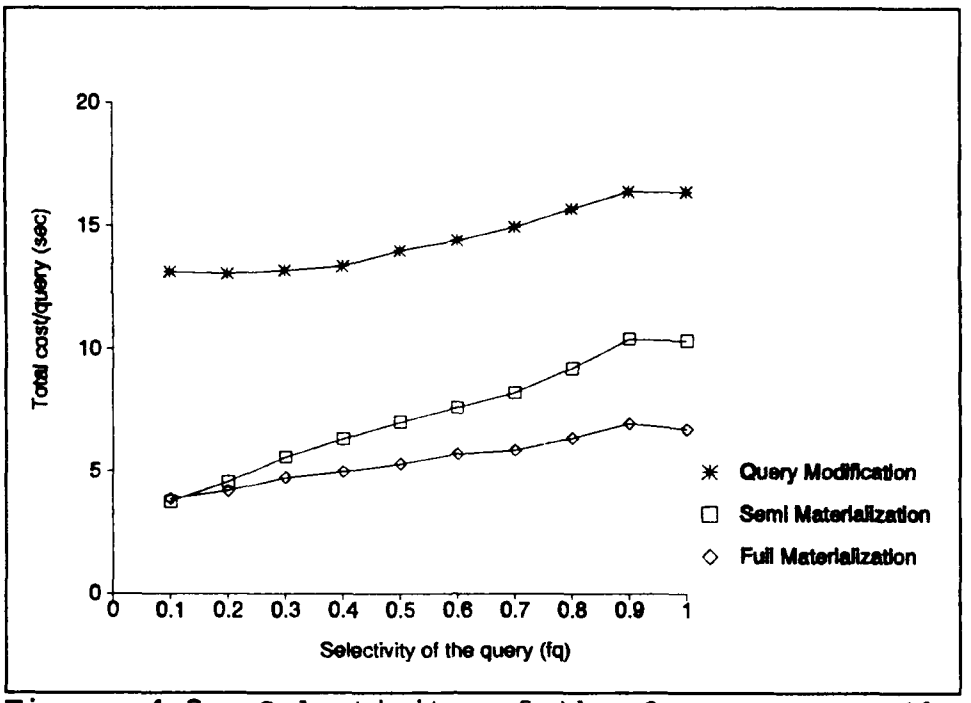


Figure 4.3 Selectivity of the Query versus the Cost-Per-Query for Model-1a.

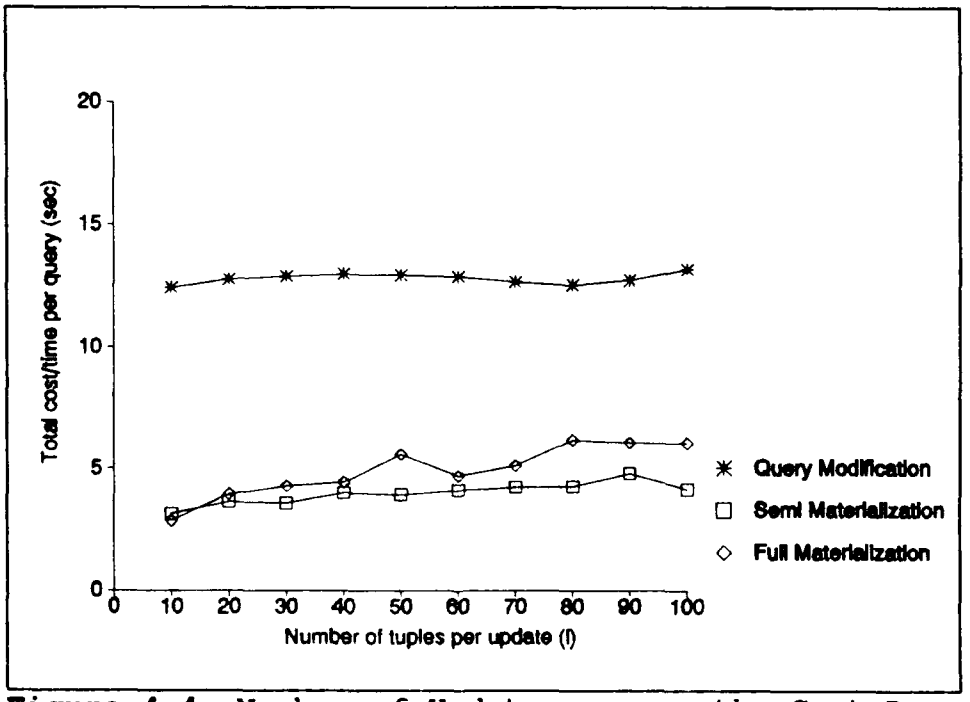


Figure 4.4 Number of Updates versus the Cost-Per-Query for Model-1a.

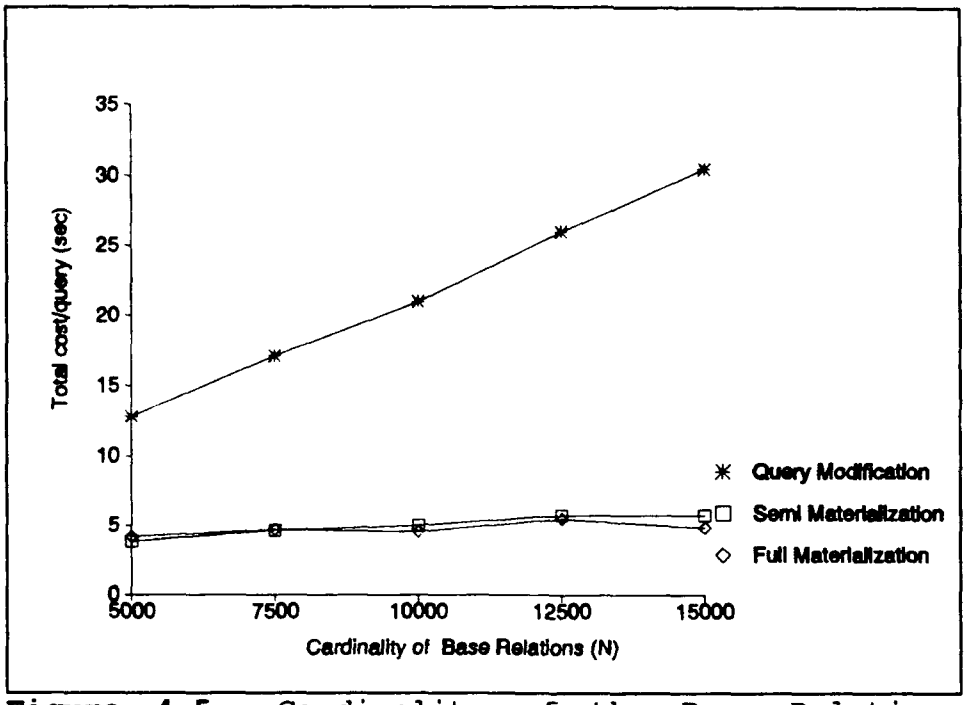


Figure 4.5 Cardinality of the Base Relations versus the Cost-Per-Query for Model-1a.

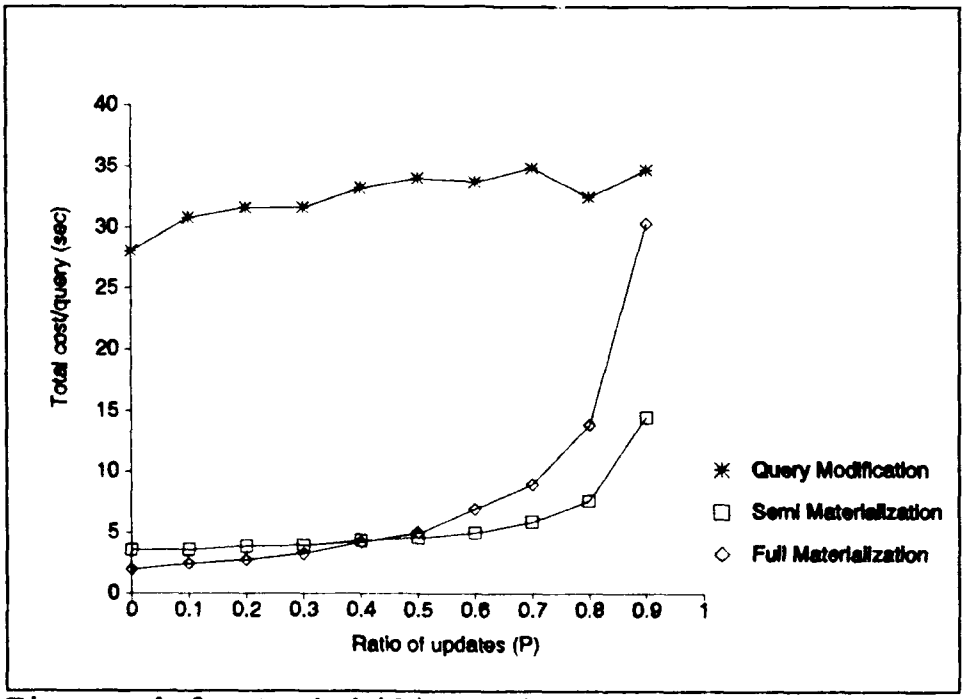


Figure 4.6 Probability of an Update versus the Cost-Per-Query for Model-1b.

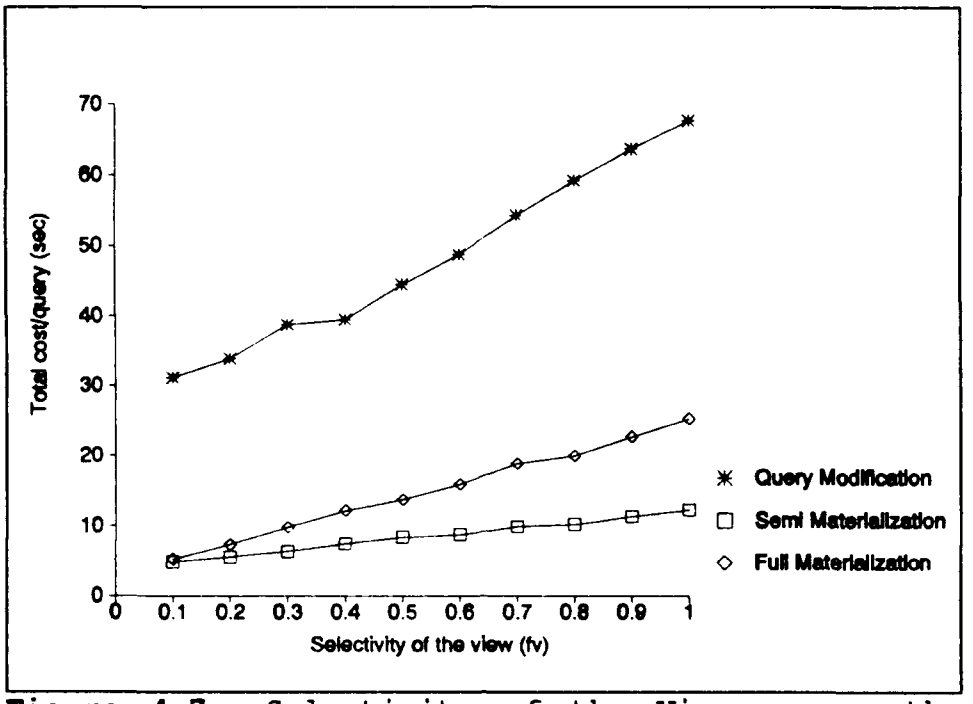


Figure 4.7 Selectivity of the View versus the Cost-Per-Query for Model-1b.

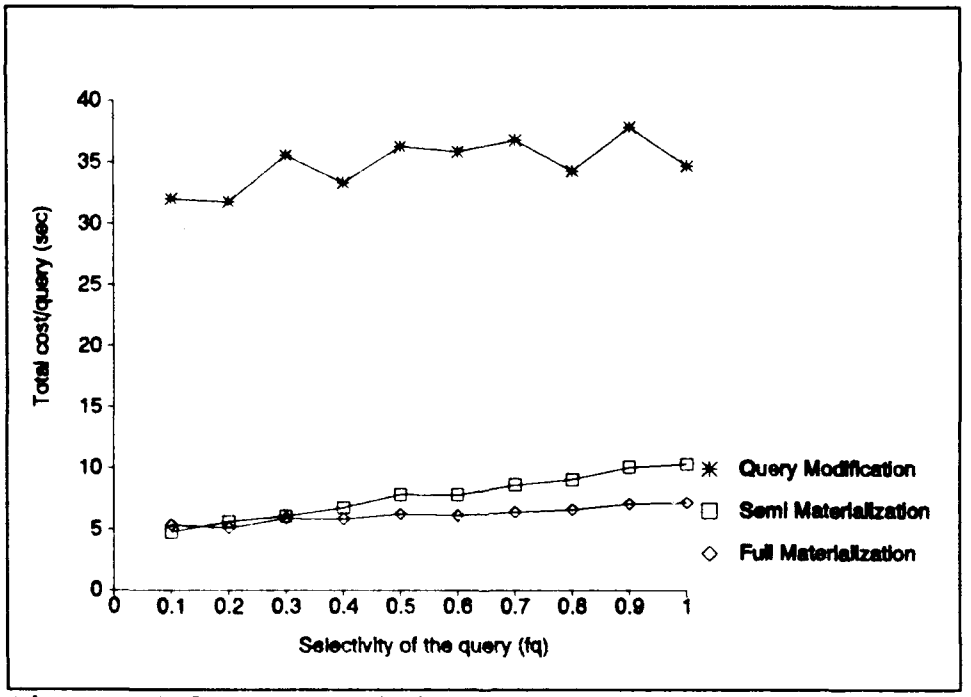


Figure 4.8 Selectivity of the Query versus the Cost-Per-Query for Model-1b.

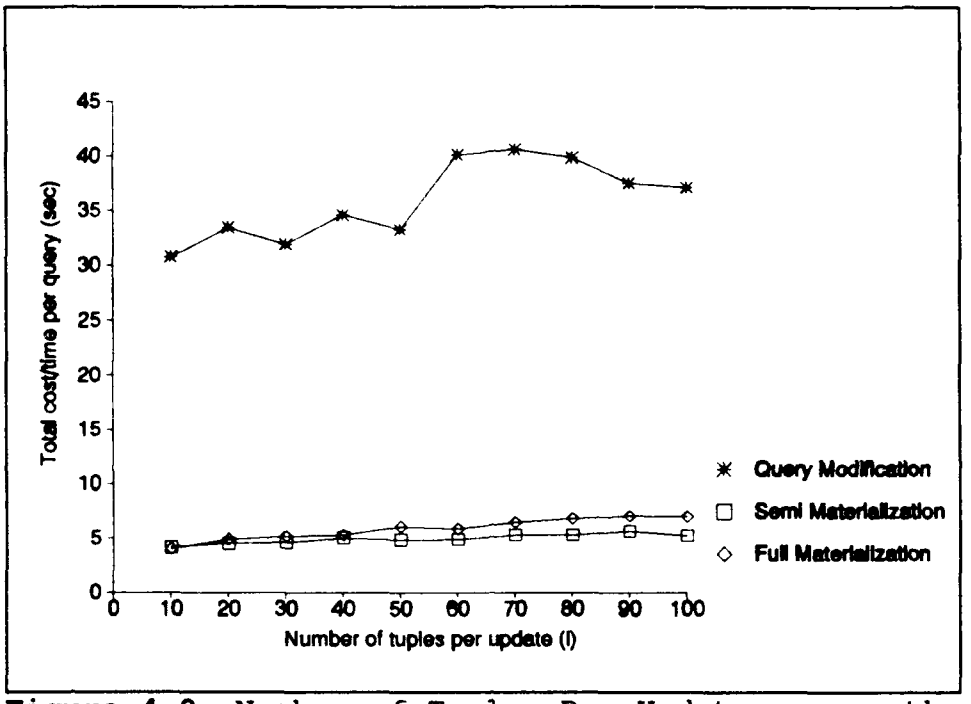


Figure 4.9 Number of Tuples Per Update versus the Cost-Per-Query for Model-1b.

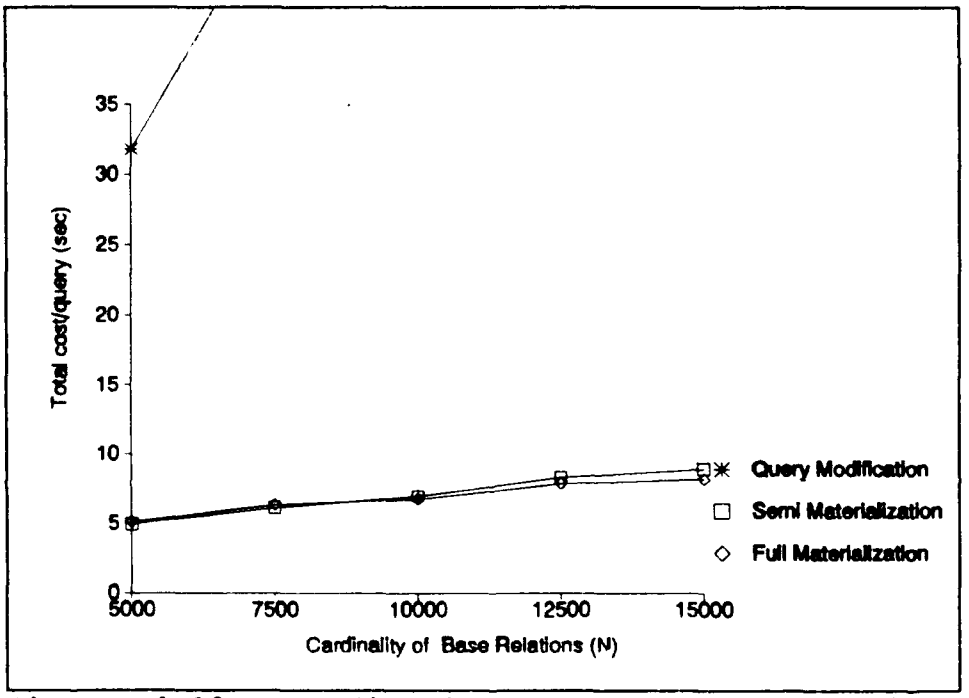


Figure 4.10 Cardinality of the Base Relations versus the Cost-Per-Query for Model-1b.

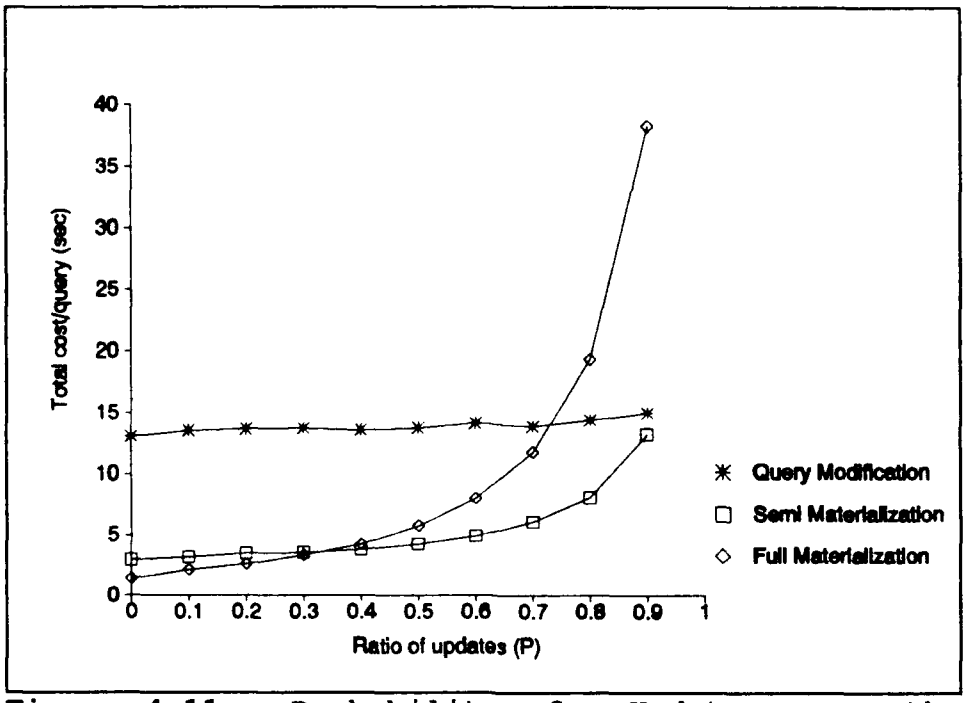


Figure 4.11 Probability of a Update versus the Cost-Per-Query for Model-1c.

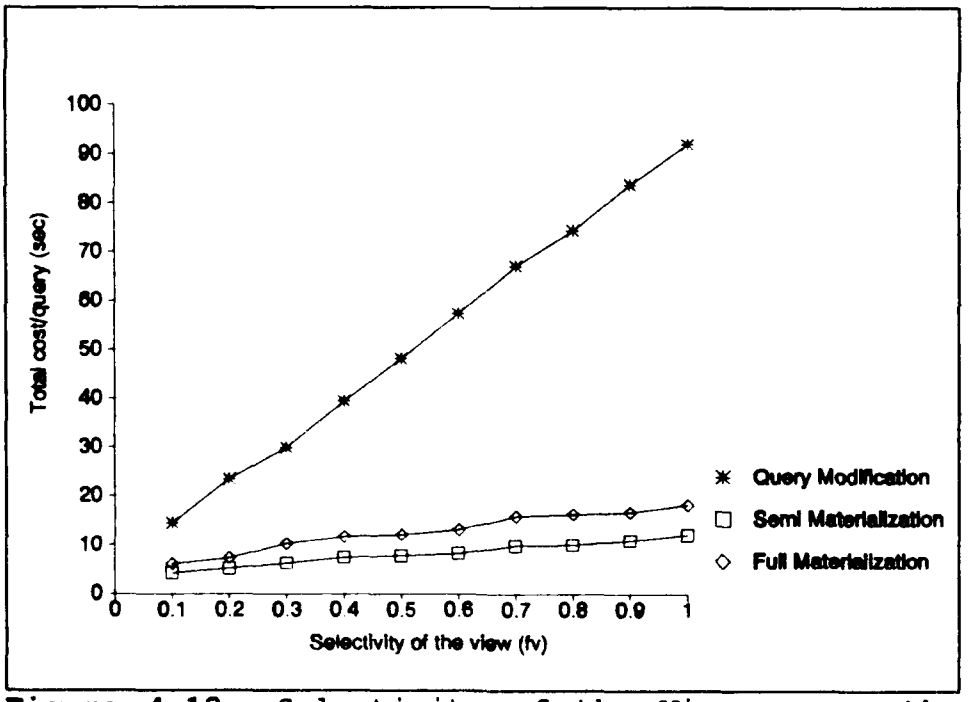


Figure 4.12 Selectivity of the View versus the Cost-Per-Query for Model-1c.

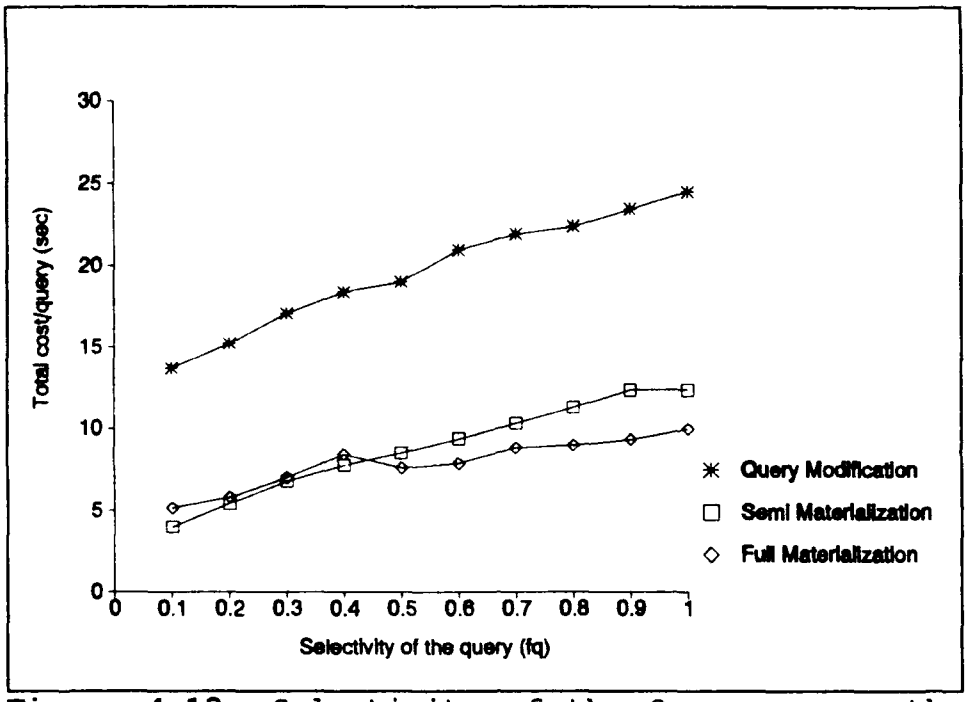


Figure 4.13 Selectivity of the Query versus the Cost-Per-Query for Model-1c.

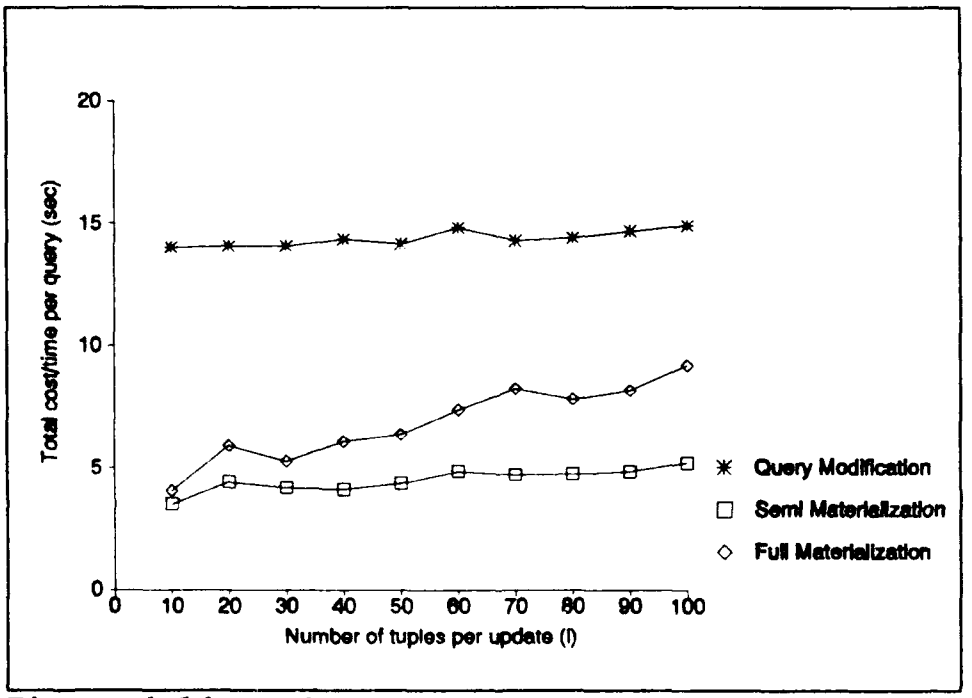


Figure 4.14 Number of Tuples Per Update versus the Cost-Per-Query for Model-1c.

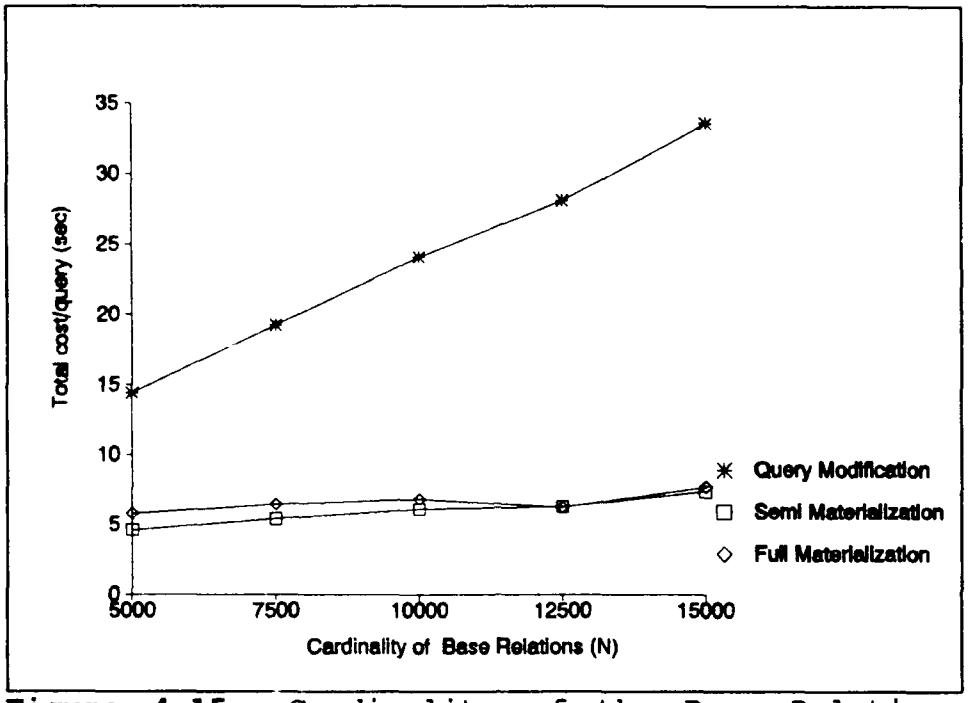


Figure 4.15 Cardinality of the Base Relations versus the Cost-Per-Query model-1c.

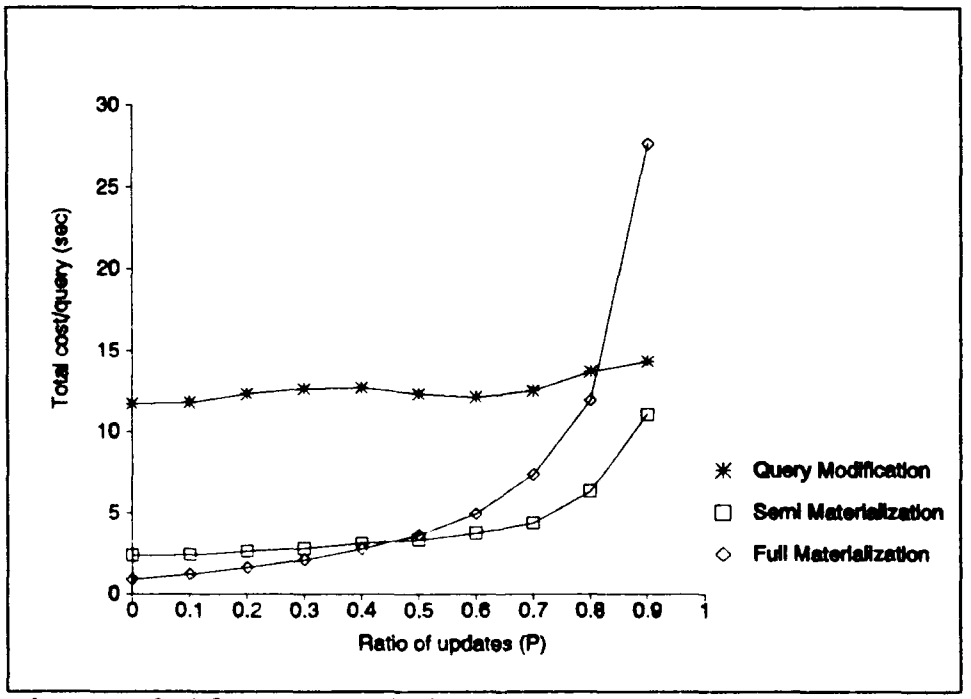


Figure 4.16 Probability of an Update versus the Cost-Per-Query for Model-2.

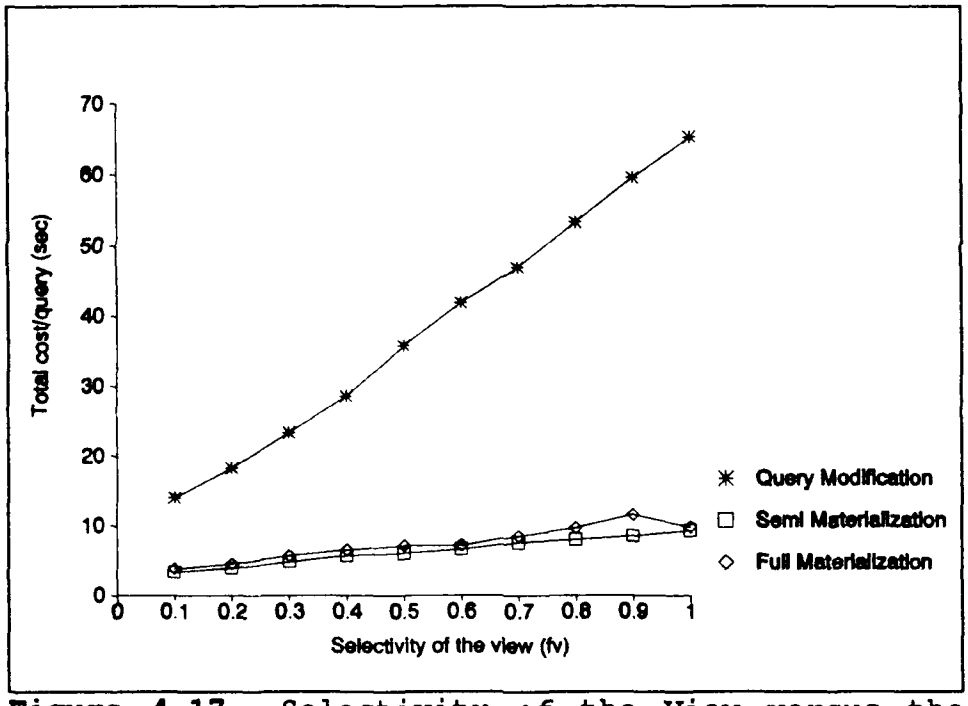


Figure 4.17 Selectivity of the View versus the Cost-Per-Query for Model-2.

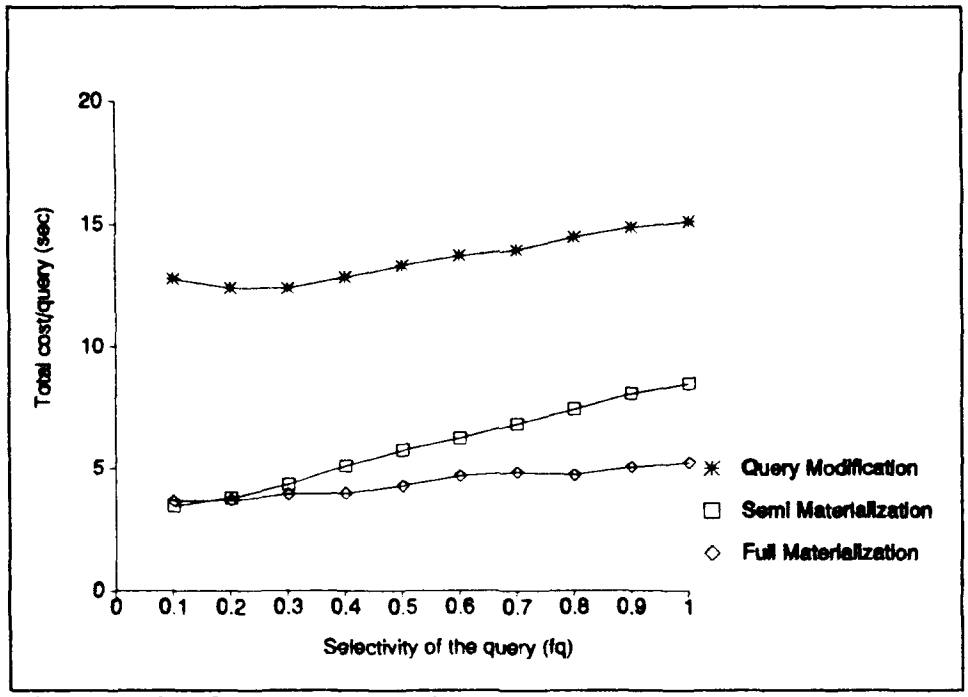


Figure 4.18 Selectivity of the Query versus the Cost-Per-Query for Model-2.

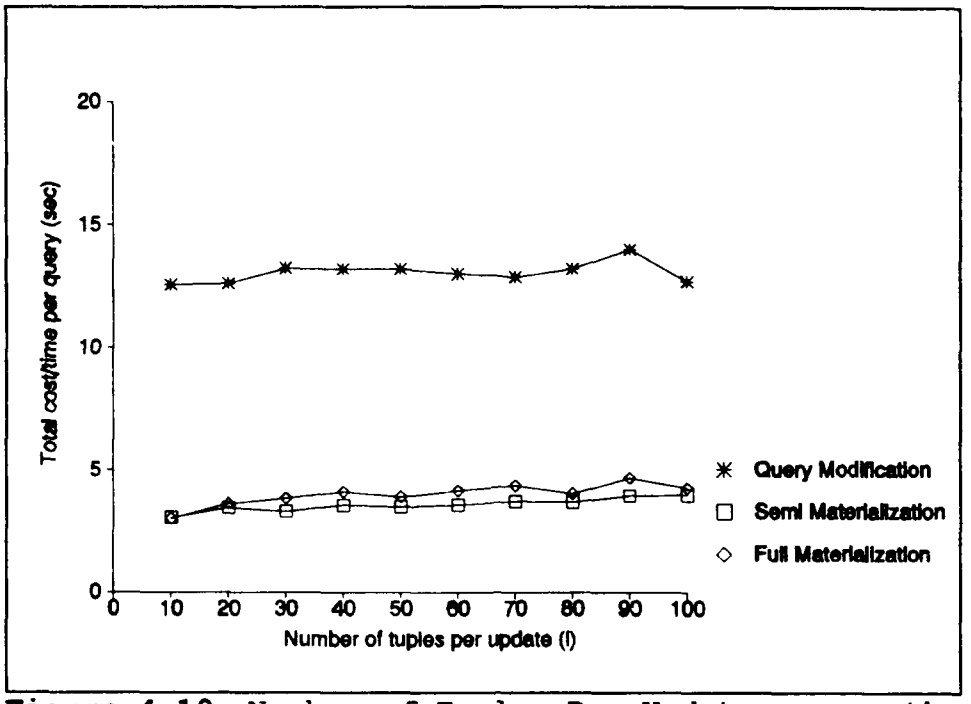


Figure 4.19 Number of Tuples Per Update versus the Cost-Per-Query for Model-2.

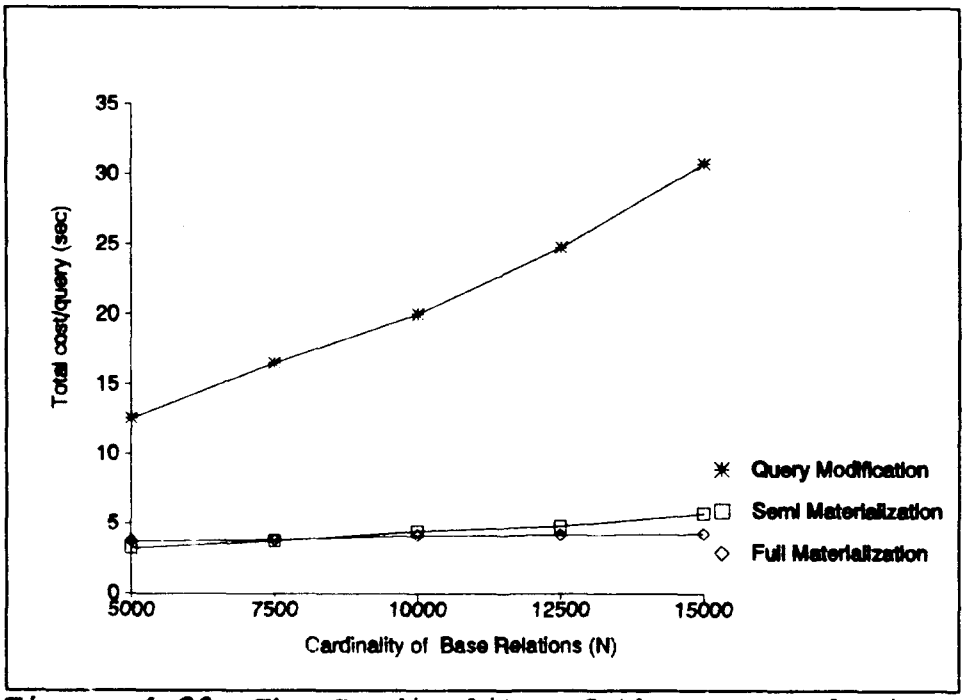


Figure 4.20 The Cardinality of the Base Relations versus the Cost-Per-Query for Model-2.

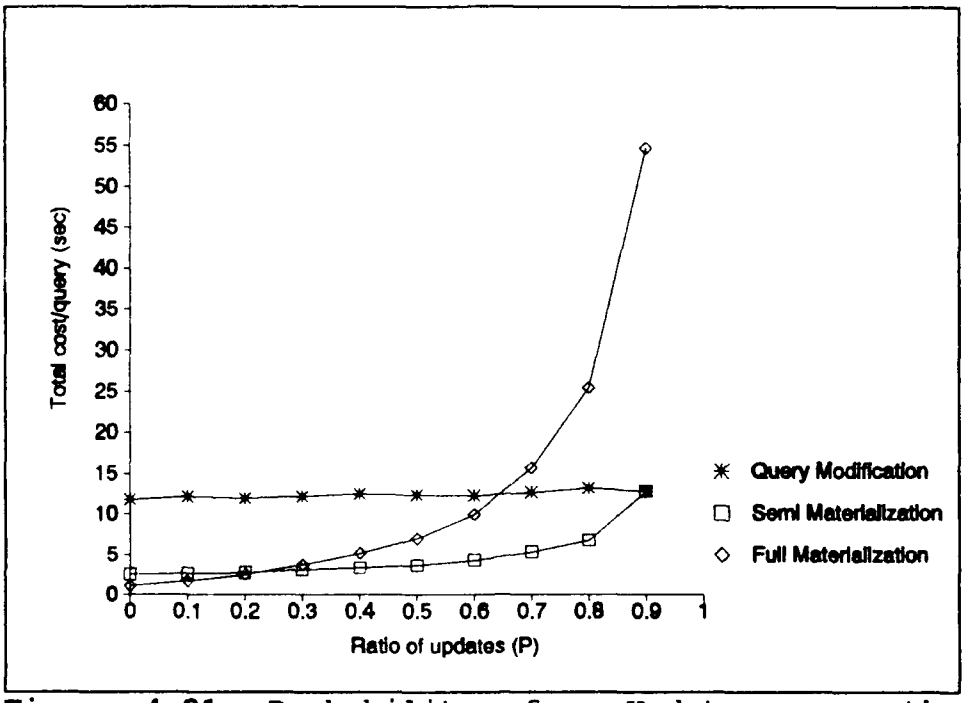


Figure 4.21 Probability of an Update versus the Cost-Per-Query for Model-3.

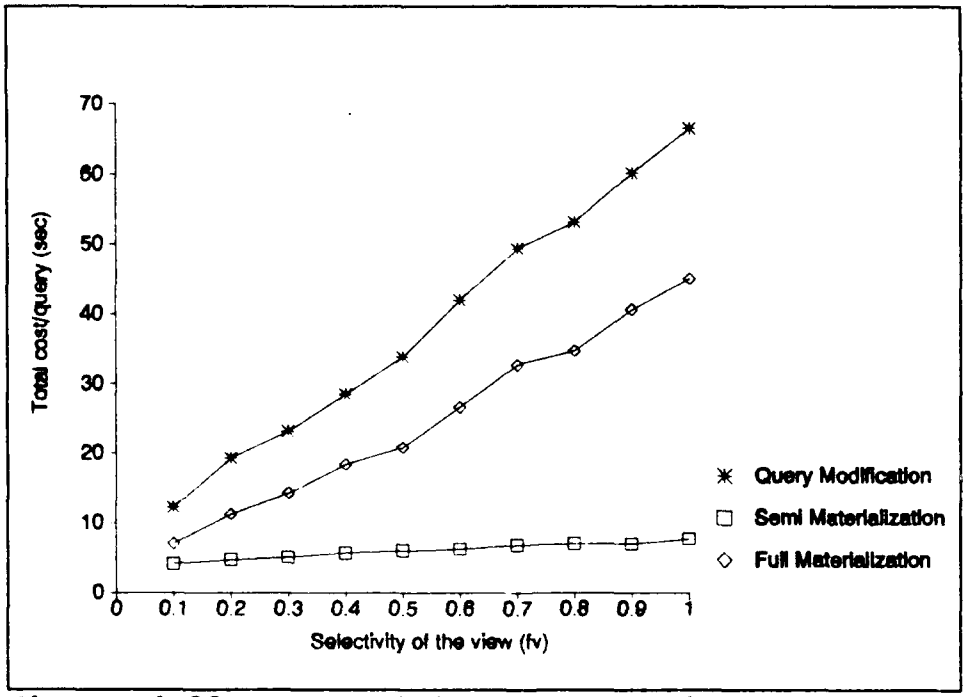


Figure 4.22 Selectivity of the View versus the Cost-Per-Query for Model-3.

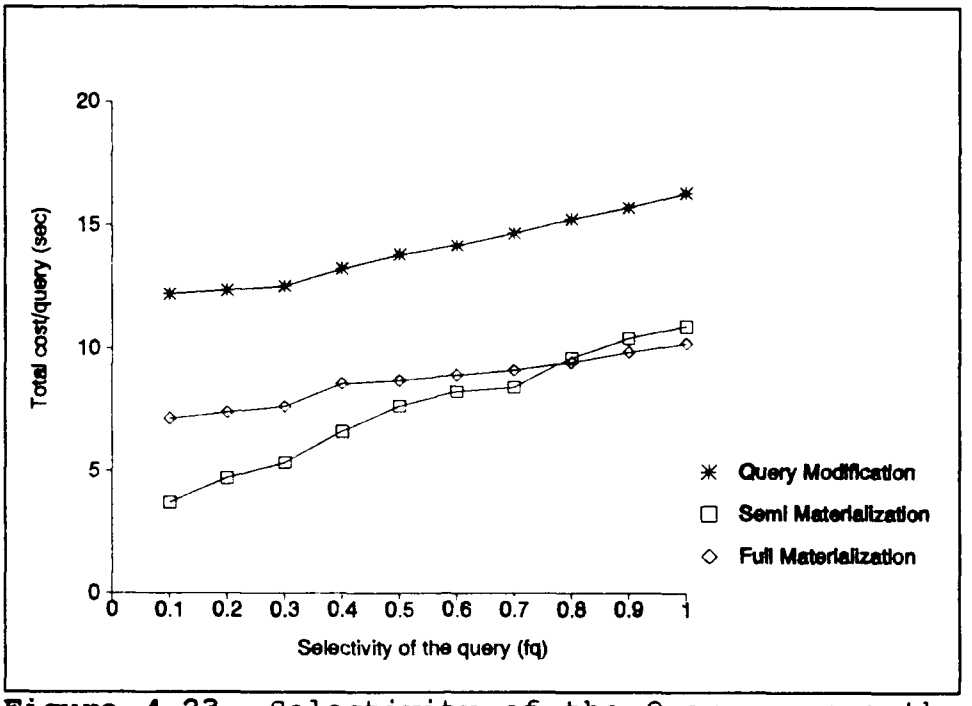


Figure 4.23 Selectivity of the Query versus the Cost-Per-Query for Model-3.

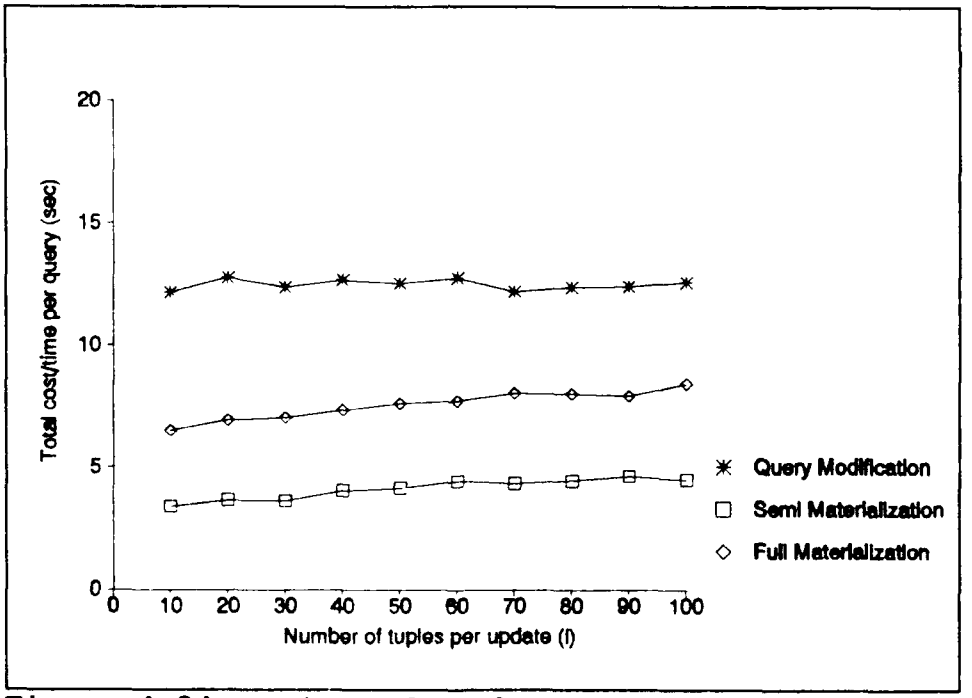


Figure 4.24 Number of Tuples Per Update versus the Cost-Per-Query for Model-3.

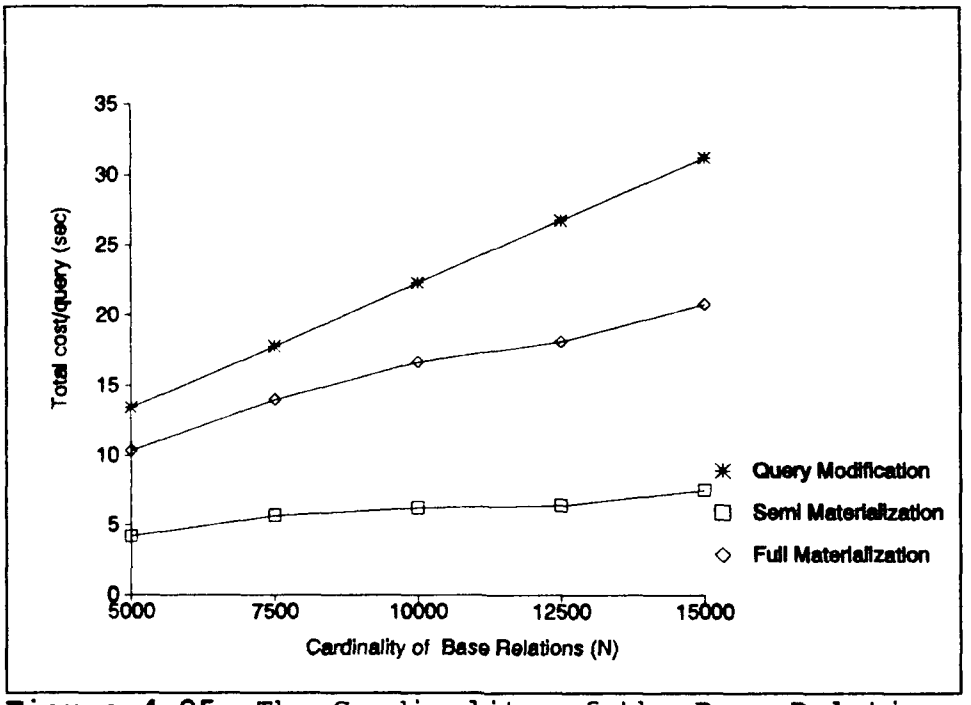


Figure 4.25 The Cardinality of the Base Relations versus the Cost-Per-Query for Model-3.

V. CONCLUSIONS AND RECOMMENDATIONS

Previous research has shown that storage of fully materialized views is a viable alternative to query modification[7]. This thesis backs the assertion that semi-materialization may perform as well or better than full materialization strategies (Kamel, 1990). While the results show the performance of the best view materialization strategy is highly application dependent, the performance trends show that semi-materialization has the potential to be a desirable option.

For select-project-join expressions, the trends of the results were surprisingly comparable for all the conditions tested. Semi-materialization and full materialization strategies performed better than query modification, except for extremely high values of P. For high values of P, f_v, l and lower values of f_q and N, semi-materialization is favored over full materialization. Under these conditions, the cost of updating the fully materialized view exceeds the benefits gained from the lower cost to perform the query. Conversely, lower values of P, f_v, l, and higher values of f_q and N favored full materialization over semi-materialization. As the cost of maintaining the materialized view was generally low and the

---

[7]Research in this area has been done by (Blakeley, 1989), (Hanson, 1987), (Srivatava, 1988), and others.

cost of scanning the redundant relations is generally high for semi-materialization. However, at low values of P, f_v, f_q, and l, the results show that the performance of both strategies is comparable. The advantages of each strategy canceled each other out (Kamel, 1990).

The analysis shows that full materialization is more sensitive to change than semi-materialization. For all three models, semi-materialization performance was relatively consistent, while full materialization performance fluctuated. In Model-2, if the condition that every tuple in R1 matches with at least one tuple in R2 is relaxed, the performance of full materialization is considerably improved relative to semi-materialization. The smaller size of the view reduces the cost of maintaining and querying the view for this model. However, it is important to note that even with this improved performance, semi-materialization is still the preferred method. In Model-3, as the cost involved in maintaining the view is high, performance for full materialization deteriorates when updates are applied to both base relations. The cost remains relatively stable for semi-materialization performance as updates in this technique are made to single relations. Full materialization updates require the screening and joining of two relations. The cost associated with these updates is dependent upon the size of the two relations. These results back the assumption that the performance of

semi-materialization would improve when updates are applied to more than one relation (Kamel, 1990).

The results presented in this thesis show that as the cardinality of the base relations, *N*, increases, the region where full materialization outperforms semi-materialization also increases. Future research is needed to further examine the performance of the three strategies over all parameter settings at higher values of *N*. Future research may also be directed towards the performance of views containing more than two base relations. The expected result, based on this thesis, is likely to be that semi-materialization will perform well as the maintenance cost for full materialization should be high.

APPENDIX A

STRUCTURE CHARTS

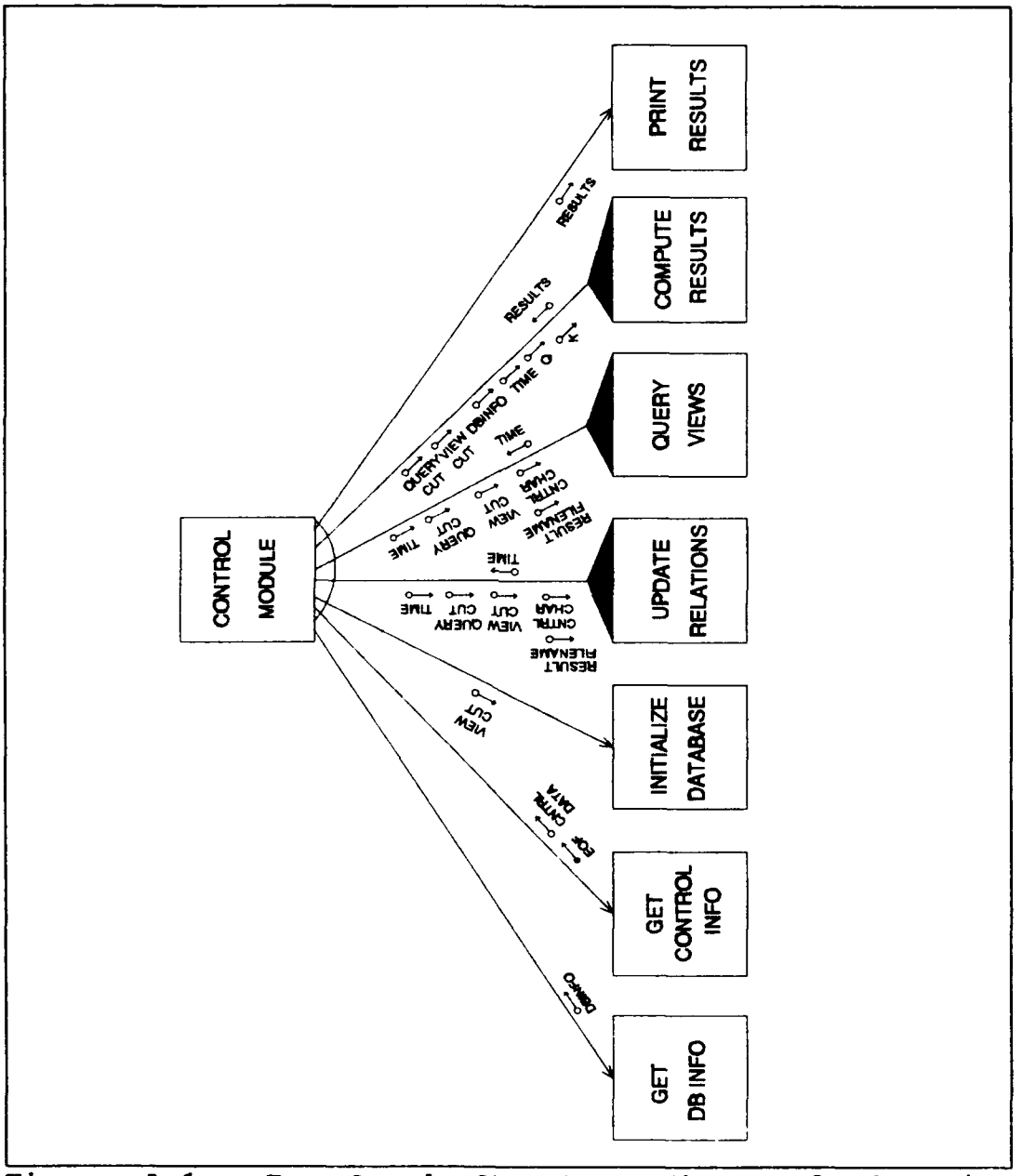


Figure A.1 Top Level Structure Chart of the View Simulation Program.

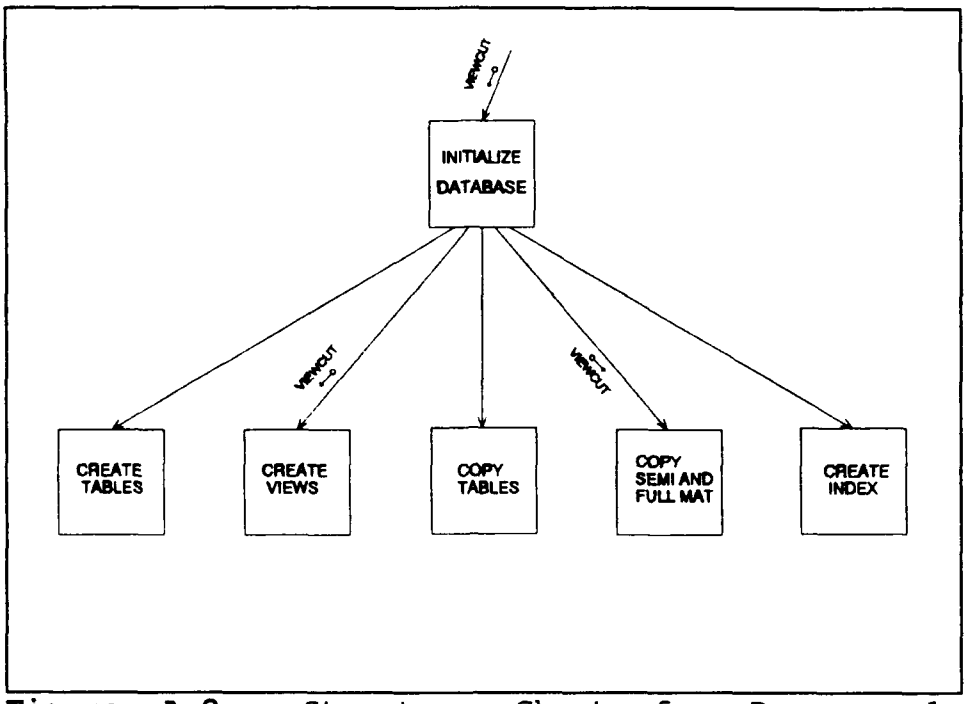


Figure A.2 Structure Chart for Process 1, Initializing the Database.

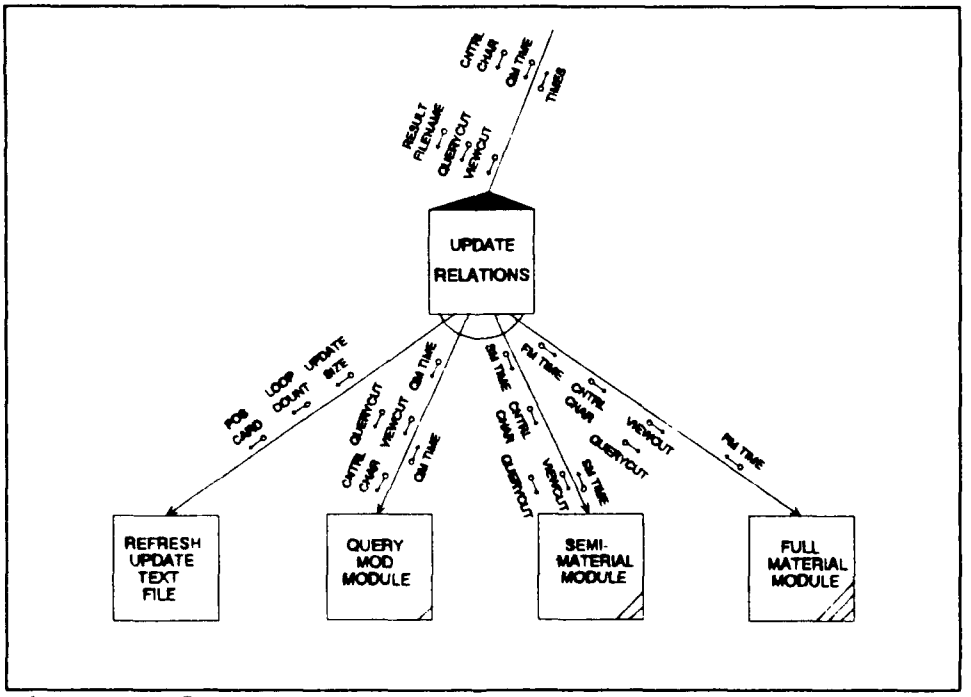


Figure A.3 Structure Chart for Process 2, Updating the Database Relations.

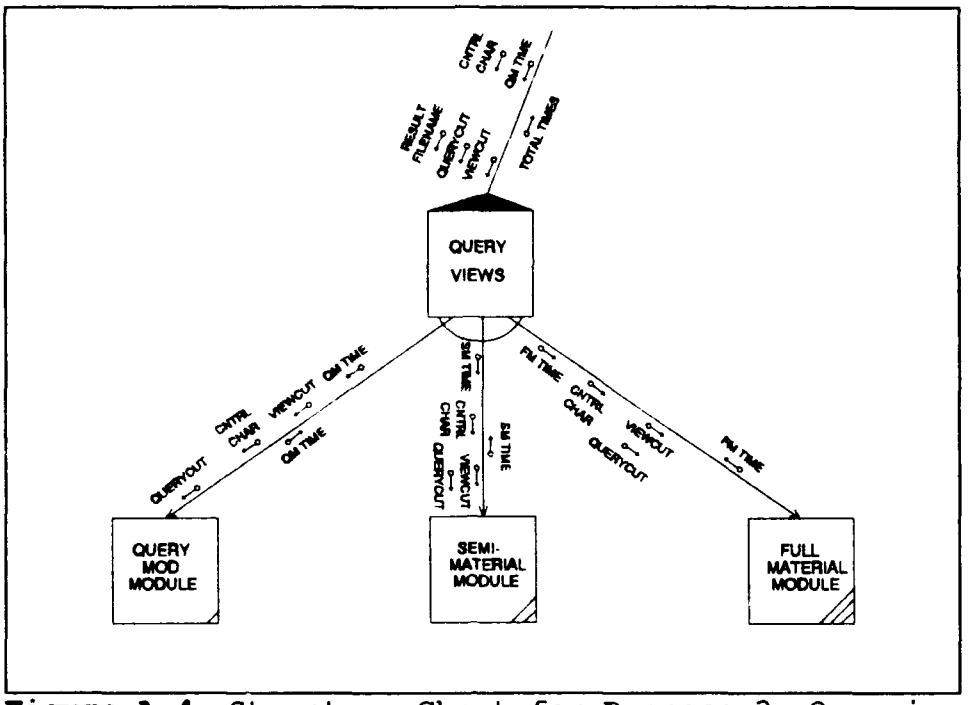


Figure A.4 Structure Chart for Process 3, Querying the Views.

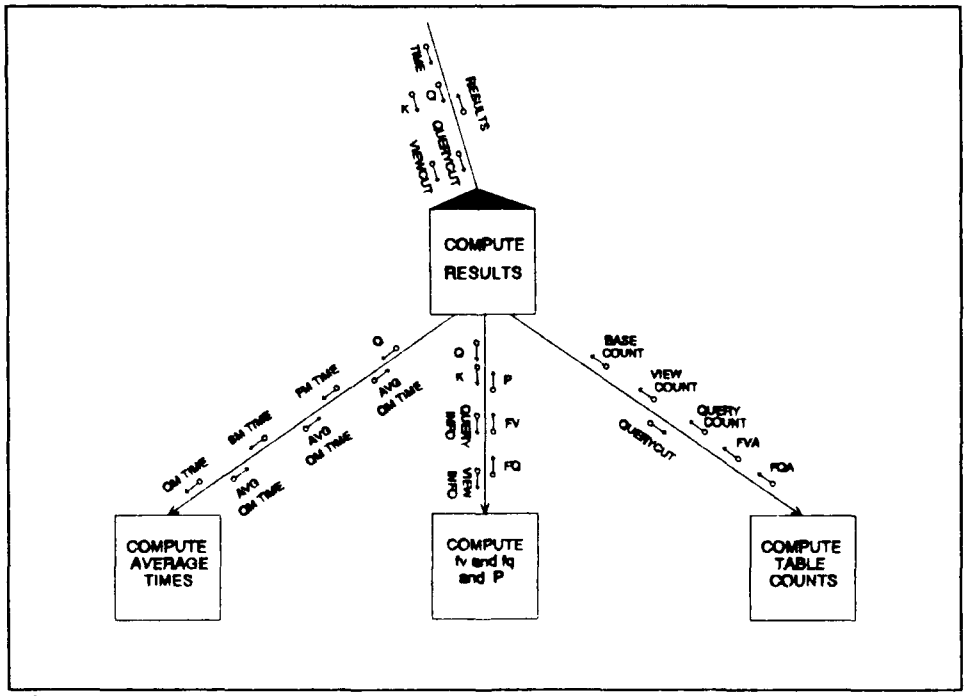


Figure A.5 Structure Chart for Process 4, Computing Results.

APPENDIX B

PROGRAM SOURCE CODE

```
/* Title              : View Materialization Simulation     */
/* Author             : Jesse T. South                      */
/* Date               : 17 June 1991                        */
/* Revised            : 08 August 1991                      */
/* Purpose            : Theses Research                     */
/* System             : IBM 80286 clone                     */
/* Compiler           : Microsoft C 6.0, INGRES precompiler */
/* Description        : The program was written as part of a*/
/*                    thesis. The purpose of this program is to*/
/*                    simulate user updates and queries on a   */
/*                    database, and to time their performance. */
/*                    The program utilizes embedded ESQL       */
/*                    command to access the INGRES relational  */
/*                    database.                                */


#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
exec sql include sqlca;


#define size 16
#define dbinfo "info.dat"
#define cntrlfl "cntrl.dat"
#define update_file "data_in"
#define finrslt "fnlrslt.dat"
#define runrslt "rnrslt.dat"

exec sql begin declare section;
#define empinfo "empdat.dat"
#define posinfo "posdat.dat"
#define skilinfo "skildat.dat"
#define updatinfo "update.dat"
exec sql end declare section;
```

```
void open_files(FILE**, FILE**, FILE**);
void close_files(FILE**, FILE**, FILE**);
void init_test_database(int);
void scan_dbinfo(long*, long*, long*, int*, int*, int*, long*,
long*, long*);
void create_tables(void);
void create_views(int);
void create_update_table(void);
void copy_base_tables(void);
void copy_semi_n_full_mats(int);
void create_table_index(void);
void module_qm(char, int, long, double*, FILE*);
void module_sm(char, int, long, double*, FILE*);
void module_fm(char, int, long, double*, FILE*);
void write_file_headings(char*, char*, FILE*, FILE*);
void write_run_result(char, char, int, long, double, long,
FILE*);
void write_final_result(int, int, long, int, long, long, long,
                float,float, float, float, float, double,
                double, double, FILE*, FILE*);
void compute_avg_time(int, double*, double*, double*);
void compute_fv_and_fq_and_P(int, int, int, int, float*, long,
                  long, long, long, float*, int, int, float*);
void compute_table_counts(long*, long*, long*, long, float*,
                            float*);
void refresh_update_text_file(long, long, long);



/* This is the Control Module, it control the entire        */
/* simulation.  The program first reads the file containing */
/* info pertaining to the database text files. It then      */
/* loops through the simulation until the end of the control*/
/* file.                                                    */

void main(void)
  {
  int K, Q, updat_siz, i, run_cnt = 0, zero = 0;
  int vmax, vbase, vincr, viewcut;
  long ecard, pcard, scard, countb, countv, countq;
  long qmax, qbase, qincr, querycut;
  float fv, fva, fq, fqa, P;
  double timeqm, timesm, timefm;
  char QUERY = 'Q', UPDATE = 'K';
  char *prm_ptr, parameter[10], *updt_ptr, updat_rel[10];
  FILE  *cntrl_fl, *fresult_fl, *run_rslt;
```

```
prm_ptr = &parameter[0];
updt_ptr = &updat_rel[0];

open_files(&run_rslt, &cntrl_fl, &fresult_fl);
scan_dbinfo(&pcard, &ecard, &scard, &vmax, &vbase, &vincr,
            &qmax, &qbase, &qincr);

while(!feof(cntrl_fl))/* while not end of file */
  {              /* Initialization */
  timeqm = timesm = timefm = 0.0;
  countb = countv = countq = 0;
  fscanf(cntrl_fl, "%d %ld %d %d %d %s %s", &viewcut,
       &querycut, &K, &Q, &updat_siz, prm_ptr, updt_ptr);
  if (run_cnt == zero) write_file_headings(prm_ptr,
                  updt_ptr, fresult_fl, run_rslt);
  init_test_database(viewcut);
  run_cnt++;
  printf("\n run # %d\n", run_cnt);
                 /* Testing */
  for(i = 0; i < K; i++)/* updates */
    {
    refresh_update_text_file(pcard, i, updat_siz);
    module_qm(UPDATE, viewcut, querycut, &timeqm, run_rslt);
    module_sm(UPDATE, viewcut, querycut, &timesm, run_rslt);
    module_fm(UPDATE, viewcut, querycut, &timefm, run_rslt);
    }
  for(i = 0; i < Q; i++)/* queries */
    {
    module_qm(QUERY, viewcut, querycut, &timeqm, run_rslt);
    module_sm(QUERY, viewcut, querycut, &timesm, run_rslt);
    module_fm(QUERY, viewcut, querycut, &timefm, run_rslt);
    }
                 /* Computation and reporting */
  compute_avg_time(Q, &timeqm, &timesm, &timefm);
  compute_fv_and_fq_and_P(vmax, vbase, vincr, viewcut, &fv,
              qmax, qbase, qincr, querycut, &fq, K, Q, &P);
  compute_table_counts(&countb, &countv, &countq, querycut,
                  &fva, &fqa);
  write_final_result(run_cnt, viewcut, querycut, updat_siz,
                  countb, countv, countq, fv, fva, fq, fqa,
                  P, timeqm, timesm, timefm, fresult_fl,
                  run_rslt);
  exec sql disconnect;
  system("rmingres");
  }

close_files(&run_rslt, &cntrl_fl, &fresult_fl);
printf("\ndisconnect complete\n");
}
```

```
/* This function combines several function modules to       */
/* initialize the test database.                            */

void init_test_database(int viewcut)
  {
  system("destroydb thesisim");
  system("createdb thesisim");
  system("addingres -B -D64000");
  exec sql whenever sqlerror stop;
  exec sql connect thesisim;
  create_tables();
  create_views(viewcut);
  copy_base_tables();
  copy_semi_n_full_mats(viewcut);
  create_table_index();
  }



/* Opens the files for the control file, and result files. */
/* If there is an error, the program will terminate         */

void open_files(FILE **run_rslt, FILE **cntrl_fl,
            FILE **fresult_fl)
  {
  *cntrl_fl = fopen(cntrlfl, "r");
  *fresult_fl = fopen(finrslt, "a");
  *run_rslt = fopen(runrslt, "a");

  if((!*run_rslt) || (!*cntrl_fl) || (!*fresult_fl))
    {
    printf("\nERROR: control or output files did not open");
    fcloseall();
    exec sql disconnect;
    exit(1);
    }
  }
```

```
/* Closes all opened files */

void close_files(FILE **run_rslt, FILE **cntrl_fl,
                 FILE **fresult_fl)
  {
  int i;

  fprintf(*fresult_fl,"\n");
  for (i=0;i<80;i++) fprintf(*fresult_fl,"*");

  fclose (*run_rslt);
  fclose (*cntrl_fl);
  fclose (*fresult_fl);
  }


/*Reads parameters used by the text generation program.     */
/*Info includess file cardinality, number of posible value  */
/* in view and query, the base value for views and query,   */
/* and the increment each increases.                        */

void scan_dbinfo(long* ecard, long* pcard, long* scard,
                 int* vmax, int* vbase, int* vincr,
                 long* qmax, long* qbase, long* qincr)
  {
  FILE* db_info;

  db_info = fopen(dbinfo, "r");

  if(!db_info)
    {
    printf("\nERROR: dbinfo file did not open ");
    fcloseall();
    exec sql disconnect;
    exit(1);
    }

  fscanf(db_info, "%ld %ld %ld\n", &*ecard, &*pcard, &*scard);
  fscanf(db_info, "%d %d %d\n", &*vmax, &*vbase, &*vincr);
  fscanf(db_info, "%ld %ld %ld", &*qmax, &*qbase, &*qincr);
  fclose(db_info);
  }
```

```
/* Creates the base and redundant tables used in the       */
/* simulation. Each table is created for each strategy so  */
/* that each strategy is tested under nearly identical     */
/* circumstances.                                          */

void create_tables()
  {

/* create query modification tables */
  exec sql create table posqm
           (e_num integer2, snum integer2, level integer1,
          keyno integer2, accinfo c86);
  exec sql create table empqm
           (e_num integer2, dnum integer2, ename c20,
            address c70, salary integer4, title c30,
            jobdesc c60);
  exec sql create table skillqm
           (snum integer2, sname c20, stype c34);

/* create semi-materialisation tables */
  exec sql create table possm
           (e_num integer2, snum integer2, level integer1,
          keyno integer2, accinfo c86);
  exec sql create table empsm
           (e_num integer2, dnum integer2, ename c20,
            address c70, salary integer4, title c30,
            jobdesc c60);
  exec sql create table skillsm
           (snum integer2, sname c20, stype c34);

  exec sql create table pos_prim
           (e_num integer2, keyno integer2);
  exec sql create table emp_prim
           (e_num integer2, ename c20, salary integer4);

/* create full materialization tables */
  exec sql create table posfm
           (e_num integer2, snum integer2, level integer1,
          keyno integer2, accinfo c86);
  exec sql create table empfm
           (e_num integer2, dnum integer2, ename c20,
            address c70, salary integer4, title c30,
            jobdesc c60);
  exec sql create table skillfm
           (snum integer2, sname c20, stype c34);

  exec sql create table full_mat
           (e_num integer2, ename c20, salary integer4,
            keyno integer2);

  }
```

```
/* Creates the views used by query modification and semi-  */
/* materialization. They are used to process queries on the*/
/* view                                                    */

void create_views(int viewcut)
  {
  exec sql begin declare section;
    int view_cut;
  exec sql end declare section;
                    /* query modification view */
  view_cut = viewcut;

  exec sql create view qm_view(e_num, ename, salary, keyno) as
            select empqm.e_num, empqm.ename, empqm.salary,
               posqm.keyno
            from empqm, posqm
            where empqm.e_num = posqm.e_num
            and posqm.level >= :view_cut;
                    /* semi-materialization view */
  exec sql create view sm_view(e_num, ename, salary, keyno) as
            select emp_prim.e_num, emp_prim.ename,
                   emp_prim.salary, pos_prim.keyno
            from emp_prim, pos_prim
            where emp_prim.e_num = pos_prim.e_num;
  }



/* Create the table used to insert records into the relation*/
/* being updated                                           */

void create_update_table()
  {
  exec sql create table update_tbl
            (e_num integer2, snum integer2, level integer1,
           keyno integer2, accinfo c86);
  exec sql copy table update_tbl
            (e_num = c0colon, snum= c0colon, level = c0colon,
             keyno = c0colon, accinfo = c0nl)
             from :updatinfo;
  }
```

```
/* Initializes the base tables. Copies the data from text  */
/* files into the database tables                          */

void copy_base_tables()
  {
  exec sql copy table posqm
           (e_num = c0colon, snum = c0colon, level = c0colon,
          keyno = c0colon, accinfo = c0nl)
            from :posinfo;
  exec sql copy table possm
           (e_num = c0colon, snum = c0colon, level = c0colon,
          keyno = c0colon, accinfo = c0nl)
            from :posinfo;
  exec sql copy table posfm
           (e_num = c0colon, snum = c0colon, level = c0colon,
          keyno = c0colon, accinfo = c0nl)
            from :posinfo;

  exec sql copy table empqm
           (e_num = c0colon, dnum = c0colon, ename = c0colon,
          address = c0colon, salary = c0colon,
            title = c0colon, jobdesc = c0nl)
            from :empinfo;
  exec sql copy table empsm
           (e_num = c0colon, dnum = c0colon, ename = c0colon,
          address = c0colon, salary = c0colon,
            title = c0colon, jobdesc = c0nl)
            from :empinfo;
  exec sql copy table empfm
           (e_num = c0colon, dnum = c0colon, ename = c0colon,
          address = c0colon, salary = c0colon,
            title = c0colon, jobdesc = c0nl)
            from :empinfo;

  exec sql copy table skillqm
           (snum = c0colon, sname = c0colon, stype = c0nl)
            from :skilinfo;
  exec sql copy table skillsm
           (snum = c0colon, sname = c0colon, stype = c0nl)
            from :skilinfo;
  exec sql copy table skillfm
           (snum = c0colon, sname = c0colon, stype = c0nl)
            from :skilinfo;
  }
```

```
/* Initializes semi and full materialization tables by     */
/* inserting records from the base relations that meet the */
/* view definition                                         */

void copy_semi_n_full_mats(int viewcut)
  {
  exec sql begin declare section;
  int view_cut;
  exec sql end declare section;

  view_cut = viewcut;

  exec sql insert into pos_prim (e_num, keyno)
            select e_num, keyno
            from possm
            where level >= :view_cut;
  exec sql insert into emp_prim (e_num, ename, salary)
            select e_num, ename, salary
            from empsm;
  exec sql insert into full_mat (e_num, ename, salary, keyno)
            select empfm.e_num, empfm.ename, empfm.salary,
               posfm.keyno
            from empfm, posfm
            where empfm.e_num = posfm.e_num
            and posfm.level >= :view_cut;

  }



/* Modifies the storage structure on field indicated and a */
/* secondary index on the fields indicated                 */

void create_table_index()
  {
  exec sql modify empqm to btree on e_num;
  exec sql modify empsm to btree on e_num;
  exec sql modify empfm to btree on e_num;

  exec sql modify posqm to btree on level;
  exec sql modify possm to btree on level;
  exec sql modify posfm to btree on level;

  exec sql modify emp_prim to btree on salary;
  exec sql modify pos_prim to btree on e_num;

  exec sql modify full_mat to btree on salary;
```

```
      /*  create secondary indexes  */
/*  exec sql create index empqmdx  */
/*            on empqm (salary);    */
/*  exec sql create index empsmdx  */
/*            on empsm (salary);    */
/*  exec sql create index empfmdx  */
/*            on empfm (salary);    */

/*  exec sql create index posqmdx  */
/*            on posqm (level);     */
/*  exec sql create index possmdx  */
/*            on possm (level);     */
/*  exec sql create index posfmdx  */
/*            on posfm (level);     */

/*  exec sql create index e_primdx */
/*            on emp_prim (salary); */
/*  exec sql create index p_primdx */
/*            on pos_prim(e_num);   */

/*  exec sql create index f_matdx  */
/*            on full_mat (salary); */
  }


/* This function simulates either an update or a query to a */
/* database relation using the query modification           */
/* methodology. Time is only accumulated for queries here.  */

void module_qm(char cntrl_char, int viewcut, long querycut,
               double *timeqm, FILE *run_rslt)
  {
  clock_t tstart = 0, tstop = 0;
  double elap_time;
  long tbl_cnt = 0;

  exec sql begin declare section;
    int view_cut;
    long query_cut;
    long qnum;
    char qname[21];
    long qkeyno;
  exec sql end declare section;

  exec sql declare qm_cl cursor for  /*make changes here to*/
  select e_num, ename, keyno         /* change query       */
    from qm_view
    where salary >= :query_cut;


  view_cut = viewcut;
```

```
query_cut = querycut;

switch(cntrl_char)  /*if K do update,if Q do query*/
  {
  case 'K':       /* updates */
    create_update_table();
    exec sql insert into posqm
             select *
               from update_tbl;
    exec sql drop update_tbl;
    break;

  case 'Q':       /* queries */
    tstart = clock();
    exec sql open qm_cl;
    exec sql whenever not found goto closeqm_cl;
    while(sqlca.sqlcode == 0)
      {
      exec sql fetch qm_cl
               into :qnum, :qname, :qkeyno;
    /*  printf("\nnumber = %d", qnum); */
      tbl_cnt++;
      }
    closeqm_cl:
      exec sql whenever not found continue;
      tstop = clock();
      exec sql close qm_cl;
    break;

  default:
    printf("\nIncorrect control character\n");
    break;
  }
             /* compute new totals */
elap_time = (tstop - tstart)/(double)CLK_TCK;
*timeqm = *timeqm + elap_time;

write_run_result('q', cntrl_char, viewcut, querycut,
                 elap_time, tbl_cnt, run_rslt);
}
```

```
/* This function simulates either an update or a query to */
/* a database relation using the semi-materialization     */
/* methodology. Accumulated time is gathered for both     */
/* updates of redundant relations and queries on the view.*/

void module_sm(char cntrl_char, int viewcut, long querycut,
          double *timesm, FILE *run_rslt)
  {
  clock_t tstart = 0, tstop = 0;
  double elap_time;
  long tbl_cnt = 0;

  exec sql begin declare section;
    int view_cut;
    long query_cut;
    long snum;
    char sname[21];
    long skeyno;
  exec sql end declare section;

  exec sql declare sm_cl cursor for  /* make changes to */
  select e_num, ename, keyno         /* to query here */
    from sm_view
    where salary >= :query_cut;


  view_cut = viewcut;
  query_cut = querycut;

  switch(cntrl_char)   /* if K do update, if Q do query */
    {
    case 'K':       /* updates */
      create_update_table();
      exec sql insert into possm
               select *
               from update_tbl;
      tstart = clock();
      exec sql insert into pos_prim
               select e_num, keyno
                 from update_tbl
                 where level >= :view_cut;
      tstop = clock();
      exec sql drop update_tbl;
      break;

    case 'Q':       /* Queries */
      tstart = clock();
      exec sql open sm_cl;
      exec sql whenever not found goto closesm_cl;
      while (sqlca.sqlcode == 0)
        {
```

```
      exec sql fetch sm_cl
               into :snum, :sname, :skeyno;
    /*  printf("\nsnum = %d", snum); */
      tbl_cnt++;
      }
    closesm_cl:
      exec sql whenever not found continue;
      tstop = clock();
      exec sql close sm_cl;
    break;

  default:
    printf("\nIncorrect control character\n");
    break;
  }
        /* compute new totals */
elap_time = (tstop - tstart)/(double)CLK_TCK;
*timesm = *timesm + elap_time;

write_run_result('s', cntrl_char, viewcut, querycut,
                 elap_time, tbl_cnt, run_rslt);
}
```

```
/* This function simulates either an update or query to a */
/* database relation using the full materialization       */
/* methodology. Accumulated time is gathered for the time */
/* to update the fulle materialized view and the time to  */
/* the actual query on the fully materialized view        */

void module_fm(char cntrl_char, int viewcut, long querycut,
          double *timefm, FILE *run_rslt)
  {
  clock_t tstart = 0, tstop = 0;
  double elap_time;
  long qcnt = 0;

  exec sql begin declare section;
    int view_cut;
    long query_cut;
    long tbl_cnt;
    long fnum;
    char fname[21];
    long fkeyno;
  exec sql end declare section;

  exec sql declare fm_cl cursor for /* make changes here to */
  select e_num, ename, keyno        /* change to query     */
    from full_mat
    where salary >= :query_cut;


  view_cut = viewcut;
  query_cut = querycut;

  switch(cntrl_char)  /* if K do update, if Q do query */
    {
    case 'K':           * updates */
      create_update_tal .e();
      exec sql insert into posfm
               select *
               from update_tbl;
      tstart = clock();
      exec sql insert into full_mat (e_num, ename, salary,
                                     keyno)
               select empfm.e_num, empfm.ename, empfm.salary,
                 update_tbl.keyno
                 from update_tbl, empfm
                 where update_tbl.e_num = empfm.e_num
                 and update_tbl.level >= :view_cut;
      tstop = clock();
      exec sql drop update_tbl;
      break;
```

```
  case 'Q':       /* queryies */
    tstart = clock();
    exec sql open fm_cl;
    exec sql whenever not found goto closefm_cl;
    while (sqlca.sqlcode == 0)
      {
      exec sql fetch fm_cl
               into :fnum, :fname, :fkeyno;
      /*  printf("\n fnum = %d", fnum); */
      qcnt++;
      }
    closefm_cl:
      exec sql whenever not found continue;
      tstop = clock();
      exec sql close fm_cl;
    break;

  default:
    printf("\nIncorrect control character\n");
    break;
  }
        /* compute new total times */
elap_time = (tstop - tstart)/(double)CLK_TCK;
*timefm = *timefm + elap_time;

exec sql select rowtot = count(e_num)  /* This was added */
         into :tbl_cnt                 /* to verify that */
       from full_mat                   /* the proper  #  */
         where salary >= :query_cut;   /* of update recs */
                                        /* were added     */
write_run_result('f', cntrl_char, viewcut, querycut,
                  elap_time, tbl_cnt, run_rslt);
}
```

```
/* Print out the header for each output file*/

void write_file_headings(char* param, char* updt_tbl,
                         FILE* fresult_fl, FILE* run_rslt)
  {
  time_t today_t;

  time(&today_t);
    /* heading for final result file */
  fprintf(fresult_fl,"\f\n  %s  -  FINAL RESULTS  -\n",
        ctime(&today_t));
  fprintf(fresult_fl,"\n The %s is the parameter being
          tested", param);
  fprintf(fresult_fl,"\n The %s table is the table being
          updated", updt_tbl);
    /* heading for detailed result file */
  fprintf(run_rslt,"\n  %s  -  RUN RESULTS\n",
          ctime(&today_t));
  fprintf(run_rslt,"\n The %s is the parameter being tested",
        param);
  fprintf(run_rslt,"\n The %s table is the table being
          updated\n", updt_tbl);
  }


/* This function is used to print out the time/cost result */
/* of each individual query or update. It is called at the */
/* end of each test module call.                           */

void write_run_result(char strat, char cntrl_char,
                int viewcut, long querycut, double elap_time,
                long tbl_cnt, FILE *run_rslt)
  {
  printf("\n%cm cc=%c vc=%d qc=%ld et=%.2lf tc=%ld", strat,
        cntrl_char, viewcut, querycut, elap_time, tbl_cnt);
  fprintf(run_rslt,"\n%cm cc=%c vc=%d qc=%ld et=%.2lf tc=%ld",
        strat, cntrl_char, viewcut, querycut, elap_time,
        tbl_cnt);
  }
```

```
/* Prints out the summary (final) results of each test run.*/

void write_final_result(int run, int viewcut,
          long querycut,int updt_siz, long countb,
          long countv, long countq, float fv, float fva,
          float fq, float fqa, float P, double timeqm,
          double timesm, double timefm, FILE *fresult_fl,
          FILE *run_rslt)
  {     /* prints to screen */
  printf("\n\nRUN# %d, VCUT= %d, QCUT= %ld, #TUP= %d,
         BASE= %ld, VIEW= %ld, QUERY= %ld", run, viewcut,
         querycut, updt_siz, countb, countv, countq);
  printf("\nFV= %.2f,    FVA= %f,    FQ= %.2f,    FQA= %f
       P= %.2f", fv, fva, fq, fqa, P);
  printf("\nTIMEQM= %.3lf sec,      TIMESM= %.3lf sec,
        TIMEFM= %.3lf sec\n", timeqm, timesm, timefm);
       /* prints to summary file */
  fprintf(fresult_fl,"\n\nRUN# %d, VCUT= %d, QCUT= %ld,
         #TUP= %d, BASE= %ld, VIEW= %ld, QUERY= %ld", run,
          viewcut, querycut, updt_siz, countb, countv,
          countq);
  fprintf(fresult_fl,"\nFV= %.2f,    FVA= %f,    FQ= %.2f,

       FQA= %f    P= %.2f",fv, fva, fq, fqa, P);
  fprintf(fresult_fl,"\nTIMEQM= %.3lf sec,      TIMESM= %.3lf
         sec, TIMEFM= %.3lf sec\n",timeqm, timesm, timefm);
       /* prints to detailed file */
  fprintf(run_rslt,"\n\nRUN# %d, VCUT= %d, QCUT= %ld,
       #TUP= %d,   BASE= %ld, VIEW= %ld, QUERY= %ld", run,
     viewcut, querycut, updt_siz, countb, countv, countq);
  fprintf(run_rslt,"\nFV= %.2f,    FVA= %f,    FQ= %.2f,
        FQA= %f  P= %.2f", fv, fva, fq, fqa, P);
  fprintf(run_rslt,"\nTIMEQM= %.3lf sec,      TIMESM= %.3lf
         sec,  TIMEFM= %.3lf sec\n",timeqm, timesm, timefm);
  }
```

```
/* Computes the average cost to query the databases for    */
/* each run.                                                */

void compute_avg_time(int Q, double *timeqm, double *timesm,
                      double *timefm)
  {
  if(Q > 0)
    {
    *timeqm = *timeqm / (double)Q;
    *timesm = *timesm / (double)Q;
    *timefm = *timefm / (double)Q;
    }
  else
    {
    printf("\n\nERROR: dividing times by 0,  **** results are
             VOID   ****\n");
    }
  }



/* Computes the intended selectivity of the view (fv),     */
/* selectivity of the query, and probability an update is a */
/* query.                                                   */

void compute_fv_and_fq_and_P(int vmax, int vbase, int vincr,
        int vcut, float *fv,long qmax, long qbase, long qincr,
          long qcut, float *fq, int K, int Q, float *P)
  {
  *fv = (float)(vmax)-((float)(vcut - vbase)/(float)(vincr));
  *fv = (*fv + (float)(vincr)/(float)(vincr))/(float)(vmax);
  *fq = (float)(qmax)-((float)(qcut - qbase)/(float)(qincr));
  *fq = (*fq + (float)(qincr)/(float)(qincr))/(float)(qmax);
  *P  = (float)(K)/(float)(K + Q);
  }
```

```
/* This function counts the number of records in the base, */
/* view, and query. It then used those values to determine */
/* the actual values of the selectivity of the view, and   */
/* query.                                                  */
void compute_table_counts(long *countb, long *countv,
         long *countq, long querycut, float *fva, float *fqa)
  {
  exec sql begin declare section;
    long query_cut;
    long tbl_cnt;
  exec sql end declare section;

  query_cut = querycut;

  exec sql create table base_mat
           (e_num integer2, ename c20, salary integer4,
           keyno integer2);

  exec sql insert into base_mat (e_num, ename, salary, keyno)
             select empfm.e_num, empfm.ename, empfm.salary,
               posfm.keyno
             from empfm, posfm
             where empfm.e_num = posfm.e_num;

  exec sql select rowtot = count(e_num)    /* coun # in base*/
           into :tbl_cnt
           from base_mat;
  *countb = tbl_cnt;

  exec sql select rowtot = count(e_num)   /* count # in view */
           into :tbl_cnt
           from full_mat;
  *countv = tbl_cnt;

  exec sql select rowtot = count(e_num) /* count # in query */
           into :tbl_cnt
           from full_mat
           where salary >= :query_cut;
  *countq = tbl_cnt;
           /* determine actual selectivities */
  *fva = (float)((double)*countv / (double)*countb);
  *fqa = (float)((double)*countq / (double)*countv);

  exec sql drop base_mat;
  }
```

```
/* The purpose of this function is to read the parameter   */
/* for the data generation program that are to be used to  */
/* build the update table. Between each run, the program   */
/* reads the parameters and refreshes them. Once refreshed */
/* the parameters are written back to file and the data    */
/* generation program is executed.                         */

void refresh_update_text_file(long card, long i,
                              long update_siz)
  {
  long update_base;
  int num_of_fields, j, change_field = 4;
  char file_name[size] = updatinfo, *file_ptr;
  FILE *updat_fl;

  struct field_attrib
    {
    char  field_type;
    int   field_width;
    char  field_info;
    long  lower_bound;
    int   increment;
    long  upper_bound;
    struct field_attrib  *next;
     };

  struct field_attrib *first_field = NULL;
  struct field_attrib *current_field = NULL;
  struct field_attrib *print_ptr = NULL;


  file_ptr = &file_name[0];
  update_base = (i * update_siz) + card +1;
              /* compute new key base number */


 /**  Read old control input for data generation program  **/

  updat_fl = fopen(update_file, "r");
  if(!updat_fl)
    {
    printf("\nERROR: update control file did not open to
            read");
    fcloseall();
    exec sql disconnect;
    exit(1);
    }

  fscanf(updat_fl, "%*d\n");
  fscanf(updat_fl, "%d\n", &num_of_fields);
  fscanf(updat_fl, "%*s\n");
```

```
for (j = 1; j <= num_of_fields; j++)
  {
  if (j == 1)
    {
    first_field = (struct field_attrib*)malloc(sizeof(struct
              field_attrib));
    if (first_field == NULL) printf("\nERROR: Memory did not
        allocate!!!");
    current_field = first_field;
    }
  else
    {
    current_field->next =(struct
       field_attrib*)malloc(sizeof(struct field_attrib));
    current_field = current_field->next;
    }
  current_field->next = NULL;

  fscanf(updat_fl, "\n%c\n", &current_field->field_type);
  fscanf(updat_fl, "%d\n", &current_field->field_width);
  fscanf(updat_fl, "%c\n", &current_field->field_info);
  fscanf(updat_fl, "%ld\n", &current_field->lower_bound);
  fscanf(updat_fl, "%d\n", &current_field->increment);
  fscanf(updat_fl, "%ld\n", &current_field->upper_bound);

  if (j == change_field)  /*changing base for keyno field*/
     {
     current_field->lower_bound = update_base;
     }
}
fclose(updat_fl);


/* write updated control input for data generation program */

updat_fl = fopen(update_file, "w");
if(!updat_fl)
  {
  printf("\nERROR: update control file did not open to
          write");
  fcloseall();
  exec sql disconnect;
  exit(1);
  }

fprintf(updat_fl, "%ld\n", update_siz);
fprintf(updat_fl, "%d\n", num_of_fields);
fprintf(updat_fl, "%s", file_ptr);

print_ptr = first_field;
while(print_ptr != NULL)
```

```
  {
  fprintf(updat_fl, "\n\n%c\n", print_ptr->field_type);
  fprintf(updat_fl, "%d\n", print_ptr->field_width);
  fprintf(updat_fl, "%c\n", print_ptr->field_info);
  fprintf(updat_fl, "%ld\n", print_ptr->lower_bound);
  fprintf(updat_fl, "%d\n", print_ptr->increment);
  fprintf(updat_fl, "%ld", print_ptr->upper_bound);
  print_ptr = print_ptr->next;
  }

fclose(updat_fl);

system("thesis");
}
```

LIST OF REFERENCES

Adiba, M., and Lindsay, B.G.,"Database Snapshots," *Proceedings of the International Conference on Very Large Database*, pp. 86-91, October 1980.

Blakeley, J.A., Coburn, N., and Larson, P.,"Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates," *ACM Transactions on Database Systems*, v. 14, pp. 369-400, September 1989.

Blakeley, J.A., Larson, P., and Tompa, F.W.,"Efficiently Updating Materialized Views," *Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data*, pp. 61-71, Washington, D.C., May 1986.

Kamel, M.N., and Davidson, S.B., "Semi-Materialization: A Performance Analysis," *Proceedings of the 23rd Annual Hawaii International Conference on Systems Engineering*, pp. 393-399, January 1990.

Kamel, M.N., and Davidson, S.B.,"Semi-Materialization: A Technique for Optimizing Frequently Executed Queries," *Data & Knowledge Engineering*, pp. 101-123, v 6, 1991.

Hanson, E.N.,"A Performance Analysis of View Materialization Strategies," *Proceedings of the 1987 ACM-SIGMOD International Conference on the Management of Data*, pp. 440-453, San Francisco, CA, May 1987.

Lindsay, B.G., and others,"A Snapshot Differential Refresh Algorithm," *Proceedings of the 1986 ACM-SIGMOD Conference on the Management of Data*, pp. 53-60, Washington, D.C., May 1986.

Srivastava, J., and Rotem, D.,"Analytical Modeling of Materialized View Maintenance," *Proceedings of the 1988 ACM-SIGMOD Conference on Management of Data*, pp. 126-134, May 1988.

Stonebreaker, M.,"Implementation of Integrity Constraints and Views by Query Modification," *Proceedings of the 1975 ACM-SIGMOD Conference on Management of Data*, pp. 65-68, San Jose, CA, June 1975.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Dudley Knox Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Prof. Magdi N. Kamel, Code AS/KA Department of Administrative Sciences Naval Postgraduate School Monterey, California 93943-5000	2
4.	LCDR Rachel Griffin US NROTC Unit Morey Hall, University of Rochester Rochester, New York 14627-0016	1
5.	LT Jesse T. South, USN 3250 S. Mead Ave. Tucson, Arizona 85730	2